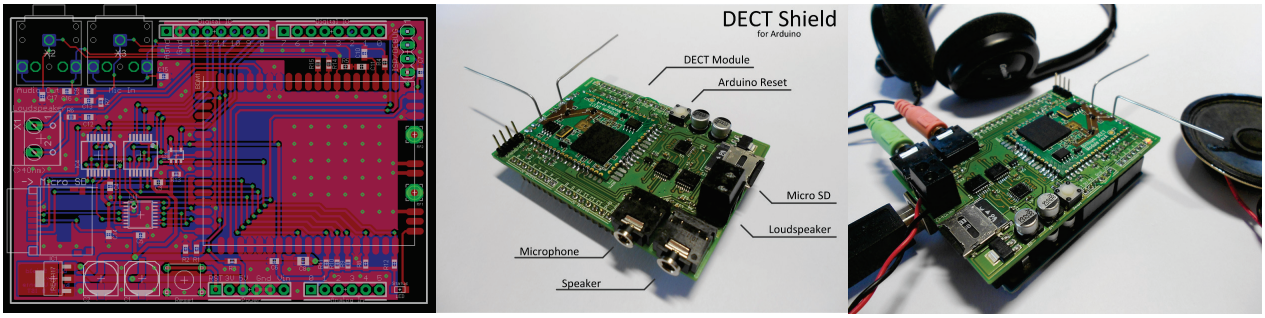
TÉCNICO LISBOA



DECT Shield for Arduino

Marcelo de Jesus Pardal Vicente

Thesis to obtain the Master of Science Degree in

Electrical and Computer Engineering

Examination Committee

Chairperson: Prof. Marcelino Bicho dos Santos

Supervisor: Prof. Maria Helena da Costa Matos Sarmento

Co-supervisor: Eng. António Manuel Pedroso Muchaxo

Member of the Committee: Prof. Mário Serafim dos Santos Nunes.

October 2012

ACKNOWLEDGEMENTS

No work of this magnitude could have been done by a single individual in a short time without support. That is why this work is not just mine, but also of every person who helped me to achieve it and therefore, they must be commended.

I would like to thank Prof. Helena Sarmento, supervisor of the project, for her constant availability to answer questions, exchange ideas and for the scientific resources that greatly aided the development and writing of this thesis.

To Bithium, I want to show my deepest gratitude for their unquestionable support, for letting me use their equipment and for having funded the necessary materials. Without their support this thesis would never have been completed on time. A special thanks to Eng. António Muchaxo, co-supervisor, for his suggestions, criticism and guidance. To Eng. Alexandre Sousa a big thanks for the knowledge he shared and patience shown when helping me. Finally, thanks to Eng. Sérgio Martins for the soldering tips.

My thanks to the INESC-ID research institution for allowing me to use their laboratories and equipment during the past months, and to Instituto Superior Técnico for its excellent education system.

I want to show my gratitude to my friends and fellow colleagues for their company and friendship. To João Marques and João Silva my thanks, for the companionship and support during the past 5 years, and a special thanks to Hugo Portela for being there when needed, during the good and bad moments.

Finally, I want to dedicate this thesis to my parents and sister for their unconditional support, dedication, aid and understanding throughout my degree. You were always there for me and I will never forget it.

To you all, a huge thanks.

Without you, this would never have been possible.

ABSTRACT

The open-source hardware development platform Arduino has been growing in recent years. Due to this growth, a wide variety of expansion boards (shields) have emerged with many different purposes in mind: from simple logic expansions to Ethernet support. Since 2011 there are shields for wireless communication, using ZigBee, WiFi and GSM, but there is not a single one using DECT (Digital Enhanced Cordless Telecommunications) technology. The main advantage of using DECT technology is the possibility to transfer audio data with high reliability up to 100 meters, without additional infrastructures or heavy processing like other technologies require (e.g. GSM).

This thesis describes the conceptualization, development and testing of a DECT shield for Arduino, that allows audio and data transmission to other DECT shields or to other DECT capable devices. An API (Application Programming Interface) in C++, compatible with the Arduino IDE, is developed in order to interface the shield using SPI (Serial Peripheral Interface). This API allows for two different programming methods: polling and events. The shield contains a microSD memory card, which is used to store audio files that can be played. The DECT shield includes three audio outputs to connect headphones, speakers and a microphone.

Tests show the shield allows using common functionalities supported by DECT (e.g. registration, audio and data transmission) without requiring a background study of DECT. Communication with the Arduino, using SPI, proved to be tolerable to errors, because the shield recovers in case of faults, and reliable since it is rare to lose messages during heavy SPI usage. Speed and range of data and audio communications are within standard DECT values. The developed API resulted in a set of simple and easy to understand functions that cover all components in the shield (e.g. speaker volume, file playback, DECT registration, etc.). A simple cordless telephone system with audio and data communication is implemented in order demonstrate the shield.

KEYWORDS:

DECT, Wireless Communications, Arduino, Audio Transmission, SPI, Micro SD card, PCB

RESUMO

A plataforma de desenvolvimento de *hardware open-source* Arduino tem vindo a crescer nos últimos anos. Devido a este crescimento, tem surgido uma vasta variedade de placas de expansão (*shields*) com as mais diversas finalidades: desde simples expansões lógicas até à inclusão de suporte Ethernet. Desde 2011, existem *shields* para comunicação sem fios, usando ZigBee, WiFi e GSM, mas ainda não existe nenhum com DECT (Digital Enhanced Cordless Telecommunications). A principal vantagem da tecnologia DECT, face às outras já mencionadas, é permitir transferir som com elevada fiabilidade até 100 metros sem necessidade de infra-estruturas ou elevado processamento adicional (e.g. GSM).

Esta tese descreve a conceptualização, o desenvolvimento e o teste de um *shield* para o Arduino, equipado com tecnologia DECT, que permite transmitir som e dados para outros *shields* semelhantes ou para dispositivos equipados com DECT. É também desenvolvida uma API (Application Programming Interface) em C++ para ser usada com o IDE (Integrated Development Environment) do Arduino a fim de fazer a interface com o *shield* usando SPI (Serial Peripheral Interface). Esta API permite usar dois modelos de programação distintos: por *polling* de eventos ou por interrupções. O *shield* também contem um cartão de memória microSD, que pode ser usado para armazenar ficheiros de som a fim de serem reproduzidos, e saídas de áudio para ligar auscultadores, colunas e microfone.

Os testes realizados mostram que os *shields* permitem usar correctamente a maioria das funcionalidades suportadas pelo DECT. A comunicação com o Arduino usando SPI mostrou ser bastante tolerável a erros e fiável, uma vez que é raro perder mensagens quando o canal SPI está congestionado. O alcance e velocidade das comunicações de som e dados ficaram dentro dos parâmetros normais do DECT. A API desenvolvida resultou num conjunto de funções muito fáceis de usar e perceber em comparação com outras APIs de *shields* existentes. A principal aplicação desenvolvida consiste num sistema de telefone sem fios com comunicação de áudio e dados, abrangendo a maioria das funcionalidades disponibilizados pelo *shield*.

PALAVRAS-CHAVE:

DECT, Comunicação sem fios, Arduino, Transmissão de som, SPI, Cartão microSD, Circuito Impresso

CONTENTS

Acknowledgements ii
Abstract iv
Resumo v
Contents vi
List of Figures viii
List of Tables x
List of Acronyms xi
1 Introduction 1
2 Technologies and Applications 3
2.1 DECT 3
2.2 Arduino 6
2.3 Arduino Wireless Shields 7
2.4 Bithium DECT Module 9
2.5 External Network Interface 10
3 Implementation 12
3.1 Requirements 12
3.2 DECT Shield 12
3.3 Hardware 15
3.3.1 Bithium DECT Module Integration 15
3.3.2 SPI Interface 16
3.3.3 Audio Connections 21
3.3.4 Power Regulation and Distribution 23
3.3.5 Arduino-Shield Interface 23
3.3.6 Circuit Boards 24
3.4 Software 28
3.4.1 Overview 28
3.4.2 Message Communication Protocol 29
3.4.3 Audio Playback 30
3.4.4 External Interface Layer 31

3.4.5 DECT API Library 33
3.5 Applications 34
3.5.1 Terminal 34
3.5.2 Text-to-speech 36
3.5.3 Telephone 37
4 Shield Evaluation 39
5 Conclusions 44
References 46
Appendix A SPI Timing Analysis 48
Appendix B Production Dossier 51
B.1 Schematic 52
B.2 Bill of Materials 54
B.3 Layout 56
Appendix C External Interface Layer 59
Appendix D DECT API 63
D.1 Library Overview 63
D.2 Library Usage 63
D.3 Module Group Functions 68
D.4 DECT Group Functions 75
D.5 Playback Group Function 82
D.6 Audio Group Functions 84
D.7 MicroSD Group Functions 86
D.8 Event Group Function 87

LIST OF FIGURES

Figure 2.1 - DECT star topology. 3
Figure 2.2 - DECT OSI Layers (from [1]). 4
Figure 2.3 - DECT frame (from [2]). 5
Figure 2.4 - Arduino Uno. 7
Figure 2.5 - Arduino wireless shields on the market. 8
Figure 2.6 - BDM layout and dimensions. 9
Figure 2.7 - BDM typical communication. 10
Figure 3.1 - DECT Shield architecture. 13
Figure 3.2 - Data communication between two DECT shields. 14
Figure 3.3 - SPI communication channels between the Arduino, BDM and microSD card. 17
Figure 3.4 - SPI communications channels in Mode A and Mode B. 17
Figure 3.5 - SPI tri-state buffer schematics diagram. 18
Figure 3.6 - SPI buffers in Mode A, BDM connected to microSD. 19
Figure 3.7 - SPI buffers in Mode B, Arduino connected to BDM. 19
Figure 3.8 - SPI buffers in Mode A, Arduino connected to microSD. 20
Figure 3.9 - BDM to speaker connections. 21
Figure 3.10 - Microphone to BDM connections. 22
Figure 3.11 - BDM to loudspeaker connections. 22
Figure 3.12 - Prototype board. 24
Figure 3.13 - DECT Shield layout (rev. A). 26
Figure 3.14 - DECT Shield assembled. 27
Figure 3.15 - DECT Shield layout (rev. B). 28
Figure 3.16 - Software abstraction layers overview. 29
Figure 3.17 - Protocol message. 30
Figure 3.18 - Interface between abstraction layers. 32
Figure 3.19 - DECT API class declaration. 33
Figure 3.20 - Terminal running on Arduino using PuTTY as interface. 35
Figure 3.21 - Telephone sketch state diagram. 37
Figure 4.1 - Bithium office Blueprint 39
Figure A.1 - SPI signals in Mode 2. 49
Figure B.1 - DECT Shield schematic (1 of 2). 52
Figure B.2 - DECT Shield schematic (2 of 2). 53
Figure B.3 - Bottom copper layer 56
Figure B.4 - Bottom solder mask 56
Figure B.5 - Top copper layer 57
Figure B.6 - Top solder mask 57
Figure B.7 - Top silkscreen layer 58

Figure B.8 - Drill plan 58
Figure C.1 - Shield registration procedure. 61
Figure D.1 - DECT API library groups. 63
Figure D.2 - DECT API library sketch flow. 64
Figure D.3 - DECT Shield empty sketch for Arduino. 64
Figure D.4 - Arduino exemplification code for data transference. 65
Figure D.5 - DECT call states sequence. 66
Figure D.6 - Arduino exemplification code for making calls. 67
Figure D.7 - Event configuration code. 87

LIST OF TABLES

Table 2.1 - DECT characteristics (from [5]). 3
Table 2.2 - Characteristics of different wireless technologies (extracted from [1] [8]). 6
Table 2.3 – Wireless shields characteristics and features. 9
Table 3.1 - SPI pins for Arduino, BDM and microSD card. 16
Table 3.2 - Shield average and peak current consumption. 23
Table 3.3 - Arduino pins for shield interface. 24
Table 3.4 - Developed sketches. 34
Table 3.5 - List of commands supported by the Terminal sketch 35
Table 4.1 - Measured average shield current consumption with a microSD card. 42
Table 4.2 - Shield cost estimation from BOM. 42
Table A.1 - Time constraints for each device in nanoseconds. 49
Table B.1 - Bill of Materials. 54
Table B.2 - Component prices from Farnell, Digikey and Mouser. 55
Table C.1 - List of commands grouped in families. 59
Table C.2 - Commands of the Module family. 59
Table C.3 - Commands of the DECT family. 60
Table C.4 - Commands of the Playback family. 61
Table C.5 - Commands of the Audio family. 62
Table C.6 - Commands of the microSD family. 62

LIST OF ACRONYMS

ADPCM	Adaptive Differential Pulse-Code Modulation
AES	Advanced Encryption Standard
API	Application programming interface
BDM	Bithium DECT Module
BOM	Bill of Materials
DCS	Dynamic Channel Selection
DSC	DECT Standard Cipher
DECT	Digital Enhanced Cordless Telecommunications
DIY	Do It Yourself
DLC	Data Link Control Layer
DRC	Design Rule Check
DSP	Digital Signal Processor
EIL	External Interface Layer
ENI	External Network Interface
FDMA	Frequency Division Multiple Access
FP	Fixed Part
GCC	GNU Compiler Collection
GMSK	Gaussian Minimum Shift Keying
I2C	Inter-Integrated Circuit
IC	Integrated Circuit
ISP	In-System Programmer
IWU	Interworking Unit
LED	Light Emitting Diode
LLME	Lower Layer Management Entity
LMP	Link Management Protocol
MAC	Media Access Control
MC	Multiple Carriers
NWK	Network Layer
PCB	Printed Circuit Board
PCM	Pulse-Code Modulation
PHY	Physical Layer
PSTN	Public Switched Telephone Network
PP	Portable Part
RF	Radio Frequency
SIM	Subscriber Identity Module
SPI	Serial Peripheral Interface
SOHO	Small Office / House Office
RFPI	Radio Fixed Part Identifier

RSSI	Radio Signal Strength Indicator
TDD	Time Division Duplex
TDMA	Time Division Multiple Access
UART	Universal Asynchronous Receiver/Transmitter
WAP	Wireless Application Protocol
WEP	Wired Equivalent Privacy

1 INTRODUCTION

Today's world thrives on wireless communications, ranging from practical Bluetooth headsets to the popular mobile phones. During the past years, the number of wireless capable devices has been growing rapidly and the range of applications has increased. Nowadays, they are used in areas which were mainly dominated by wired solutions, for example in sensor networks for home automation such as alarm systems, remote doors and wireless sound systems.

Digital Enhanced Cordless Telecommunications [1] (DECT) is the second most successfully ETSI (European Telecommunications Standard Institute) standard after GSM, dominating the wireless voice application market with a share of 73% [2]. It was created primarily for cordless phone systems in Small Office/Home Office (SOHO) applications, but currently it is also used in a wide variety of voice and data applications, such as baby monitors and traffic lights [3]. DECT standard ensures high quality voice transmission without interference up to 100 meters of distance (no obstacles), not using the highly congested 2.4 GHz spectrum, but the licensed 1.8 GHz band.

Although DECT products are widely commercialized, there are not many resources available for independent developers who wish to use it, when compared with other wireless technologies (e.g. GSM, Wi-Fi, Zigbee, Bluetooth). DECT development boards or kits are relatively expensive and hard to acquire, requiring direct contact with manufacturers.

Arduino is a low-cost open-source hardware development platform being widely used among hackers, hobbyists and scientists for small and medium size projects [4]. The key aspects of the Arduino are its simplicity, ease of use, fast prototyping and expansion capabilities. High popularity in recent years resulted in a vast number of daughter boards (shield) that add new functionalities to the Arduino (e.g. Ethernet, Wi-Fi).

The main motivation behind this thesis is the lack of low-cost DECT development resources and materials as already stated. Although there are a variety of DECT capable devices, there is not an economic and easy way to interact with them. The system implemented in this thesis tries to solve this problem by bringing DECT to the general public.

Several wireless audio applications created by independent developers and hobbyists are often limited by the technology used: Bluetooth has low range, Zigbee does not support audio natively, Wi-Fi VOIP consumes a lot of resources and GSM requires a paid carrier subscription. DECT solves some of these issues by providing better distances than Bluetooth, supporting audio natively, not consuming a vast amount of resources and not requiring a subscription with a cellular operator.

By combining the Arduino with a shield containing DECT technology, it is intended to create a cheap solution which is capable of proving DECT functionalities, such as data and audio transmission, to the public in a familiar way. Hopefully, this system will aid developers in applications, which were previously limited.

The final system should be able to serve as a fast prototyping method for application which requires audio support. Applications such as intercommunication devices and audio / data transmitters need to be easy to implement without spending much time programming or reading documentation.

The main goal of this work is to conceptualize, implement and test an Arduino shield containing a Bithium DECT Module (BDM). This includes the development of the BDM firmware code, the API for the Arduino, the printed circuit board (PCB), a set of application examples and the documentation. The specific objectives to be achieved are:

- To analyze commercially available wireless shields for Arduino.
- To analyze the Arduino and the BDM specifications.
- To specify, design, manufacture and assemble the DECT shield.
- To specify and implement the necessary software components.
- To test the shields using a set of application examples.
- To write the thesis.

This thesis is organized in five major chapters. Chapter 2 describes the technologies and devices used in the project. It also includes a description of existing wireless shields for the Arduino. The hardware and software implementation, including the decisions and the applications developed, are described and explained in detail in Chapter 3. Tests performed to verify the shield performance and the results obtained are discussed in Chapter 4. Finally, Chapter 5 contains the project conclusions and presents some suggestions for future work and improvements.

2 TECHNOLOGIES AND APPLICATIONS

2.1 DECT

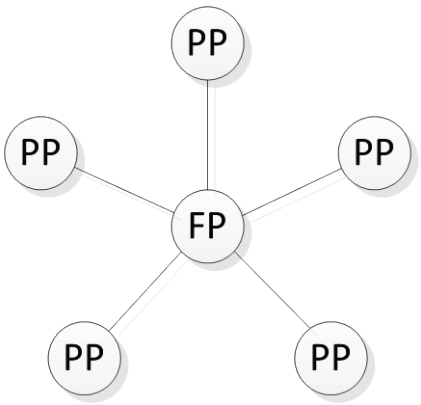
DECT [5] is a standard specified by the ETSI for wireless communications developed in Europe in 1992. Some of DECT standard characteristics are present in Table 2.1.

Table 2.1 - DECT characteristics (from [5]).

Frequency band	1880-1900MHz
Number of carriers	10
Carrier spacing	1.728 MHz
Access technique	TDMA, TDD, FDMA
Traffic duplex channels	12
Data rate per channel	32 kbps
Range	30 m or 300 m
Modulation	GMSK (BT=0.5)
Sensitivity	-86 dBm at 0.01 BER
Average RF power / slot	10 mW
Peak RF power	250 mW

DECT typically works in star topology as a micro-cell system, formed by base stations, FPs (Fixed Parts), and portable terminals, PPs (Portable Parts). In order for a cell to exist, it requires a base station and at least one portable part which the station will serve. These cells are generally connected to each other through Public Switched Telephone Networks (PSTN). DECT defines the radio interface between the portable parts and the fixed part, as represented in Figure 2.1.

Figure 2.1 - DECT star topology.

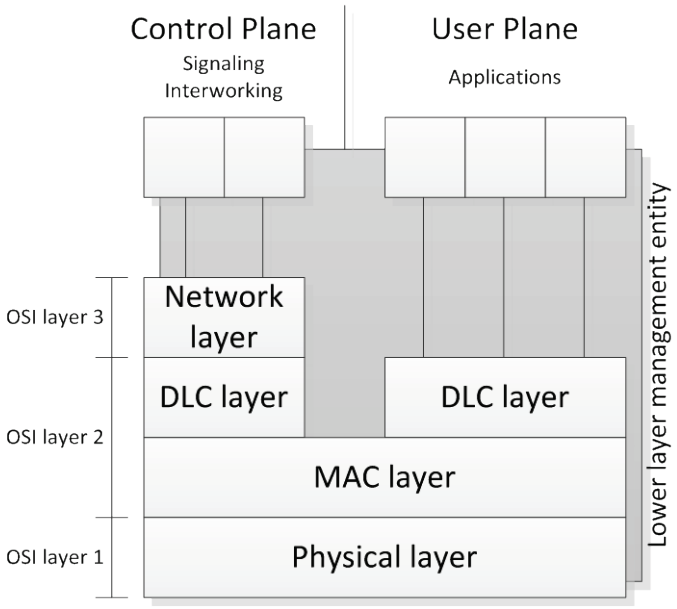


DECT relies on a method called Dynamic Channel Selection (DCS) for the channel allocation procedure [6]. Instead of using a fixed channel for communication, the portable parts are continuously sensing the available channels and trying to use the best one. This method relies on the portable parts

to select the channel instead of the fixed part. The portable terminal chooses the least interfered channel from a periodically updated list of all 120 existing channels. DECT is also capable to handover time slots in order to improve communication quality in a seamless way. This channel allocation method permits DECT to adapt automatically to the environment conditions. The DECT spectrum can be dynamically shared with other DECT applications and base stations without requiring prior channel allocation. DECT uses the licensed 1.88-1.90 GHz spectrum, therefore only DECT applications are allowed to use it.

DECT protocol architecture follows the lower layers of the OSI reference model [7], as illustrated in Figure 2.2. It has a physical layer (OSI layer 1), a data link layer (OSI layer 2) and a network layer (OSI layer 3). The data link layer is formed by a medium access control layer (MAC) and a data link control layer (DLC). The DLC layer is separated in two planes, the control plane is used for DECT signaling and interworking, and the user plane for transferring user information (e.g. voice and data). The lower layer management entity (LLME) is used for all control activities.

Figure 2.2 - DECT OSI Layers (from [1]).

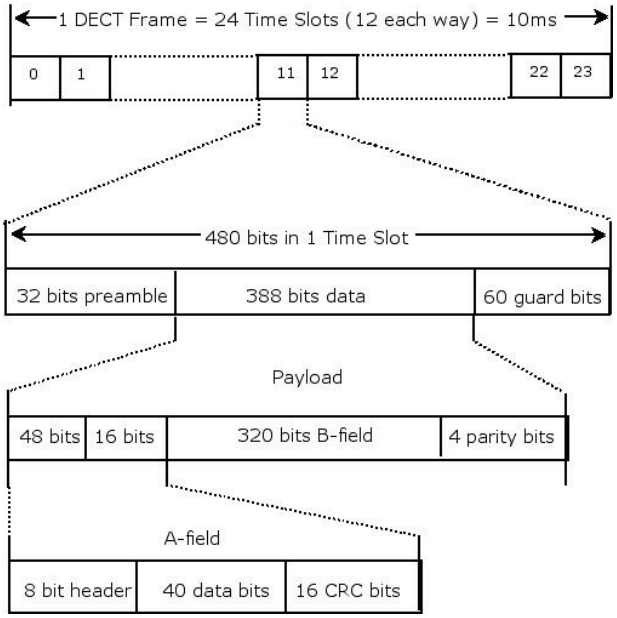


The Physical layer (PHY) is responsible for specifying the radio parameters such as frequency, timing, transmission power, bit and slot synchronization, and for the transmitter and receiver performance. It uses TDMA (Time Division Multiple Access), TDD (Time Division Duplex) and FDMA (Frequency Division Multiple Access). The DECT spectrum is split into 10 frequencies of operation, allowing multiple carriers (MC). Due to the use of FDMA, each channel will use one of ten frequencies available.

DECT employs 10ms frames, each with 24 slots (TDMA) as shown in Figure 2.3. The slots are separated into two groups of 12 slots: one group for the uplink (PP to FP) and the other for downlink (FP to PP). A duplex DECT channel is a combination of one uplink and one downlink slot through the

use of TDD. DECT supports up to 120 duplex channels as result of 10 different frequencies and 12 duplex channels per frequency.

Figure 2.3 - DECT frame (from [2]).



Time slots are formed by a 32 bit preamble marking the beginning of a slot, a 388 bit payload which contains DECT control and user information and a 60 guarding bits to prevent slot collisions.

The MAC layer specifies how logical channels are mapped and multiplexed to the physical channels. It provides channels for signaling information to the control plane. These channels are mapped to the A-field in the payload which has 64 bits. The B-field consists of 320 bits where user information is placed, resulting in a normal bit rate of 32 kbps because the slots are transmitted every 10ms. This field is mainly used to send voice over the air in ADPCM format and is not suitable for data because there are only four parity bits for error checking.

For data transmission, the 320bits in B-field are organized into four 80bits blocks with 16bits for error correction (protected B-field). Using protected B-field reduces data rate to 25.6 kbps. In order to obtain higher transfer speeds, it is possible to group multiple sequential slots.

The DLC layer delivers services for the control and user plane. The control plane provides a point-to-point service and a broadcast service taking care of addressing, frame delimiting, error control, flow control, segmentation and fragmentation of frames, and connection handover. The user plane provides frame relaying, frame switching and rate adaptation services. The DLC layer requires two bytes for data transfer when using protected B-field, reducing the data rate to 24 kbps. DECT control frames can also be used to transmit data up to 1.2 kbps.

Table 2.2 presents some characteristics of DECT, ZigBee, Bluetooth, Wi-Fi and GSM/GPRS. DECT advantages over other commercially available wireless technologies are low power consumption, short/medium communication distance, reliable voice communication but provides

smaller data rates. DECT is the only standard which provides short range voice communication natively.

Table 2.2 - Characteristics of different wireless technologies (extracted from [1] [8]).

Feature	DECT	ZigBee	Bluetooth	Wi-Fi	GSM/GPRS
Topology	Star	Mesh, Tree, Star	Star	Star	Tree
Spectrum	1.9GHz	868/915MHz, 2.4GHz	2.4GHz	2.4GHz, 5.8GHz	800MHz, 1.9GHz
Security	Pin code, DSC	AES-128bit	Pin code, LMP	WEP, WPA	SIM, 64bit
Data Rate (max)	32 kbps	250 kbps	723 kbps	105 Mbps	114 Kbps
Range (max)	300 m	1.5 km	100 m	100 m	N/A[1]
Power	Very Low[2]	Very Low	Low	High	High

2.2 ARDUINO

Arduino [9] is an open-source electronics prototyping platform base on flexible, easy-to-use hardware and software. It is intended for artists, designers, hobbyists and anyone interested in creating interactive objects or environments.

There are twenty different official prototype boards, fully supported and documented by the Arduino development team. The open-source hardware consists of an 8-bit Atmel megaAVR processor [10] with on-board input and output interfaces. The software consists of a simple IDE, with C/C++ support, to write sketches and a custom GCC compiler. The Arduino processor has a boot loader which is used to transfer new sketches, an Arduino program, through an USB or RS-232 interface.

The Arduino IDE is a multiplatform program developed in Java which is used to write, compile and deploy sketches to the Arduino. The IDE comes with several C/C++ libraries imported from the Wiring [11] project, providing a set of base functions targeted for hardware sensing and controlling.

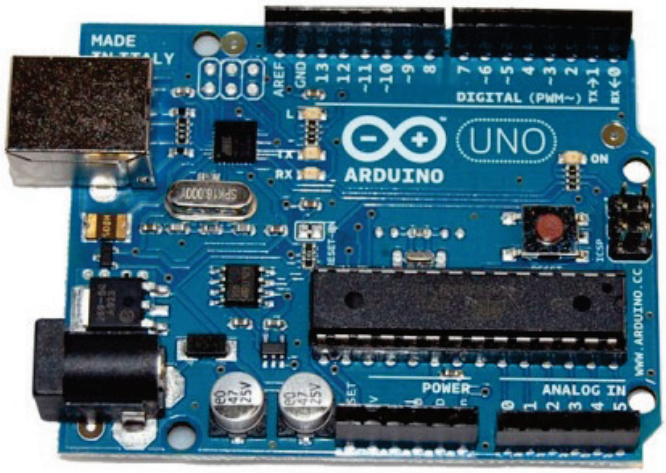
Arduino boards follow the same layout and specifications regarding pin placement, to allow the connection of daughter boards, or shields, without incompatibility issues. The Arduino shields are stackable hardware boards that connect to the Arduino using the pin headers and extend its functionalities. Nowadays, there are more than 250 available shields in the market [12].

The boards used in this thesis are Arduino Uno prototype boards [13] (Figure 2.4), with a USB connection, a DC power jack and twenty I/O ports for connectivity. Some of these I/O ports have multiple functionalities, such as UART, SPI, I2C or PWM. The processor is an 8-bit Atmel Atmega328 working at 16 MHz. It has 32 kB of flash programming space in which 512 bytes are reserved for the boot loader, 2 kB of RAM and 1 kB of EEPROM. The boards have an on-board 5 V regulator for the

[1] Depends on the infrastructure (e.g. cell towers, repeaters).
[2] DECT Ultra Low Power (ULP).

processor and I/O ports, and a 3.3 V regulator for the USB interface. Both regulators can be accessed using pin headers.

Figure 2.4 - Arduino Uno.



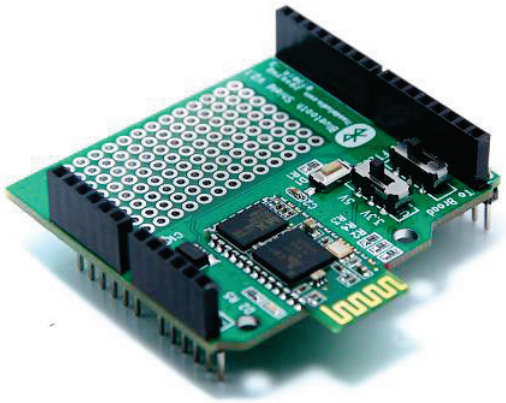
2.3 ARDUINO WIRELESS SHIELDS

There are shields in the market that already add wireless functionalities to the Arduino, from Bluetooth to GSM. This section presents an overview of some of these shields in order to compare their advantages and disadvantages.

The most popular Bluetooth shield for the Arduino is the ITead Stackable Bluetooth Shield [14] as depicted in Figure 2.5.(a). This shield has half the size of an Arduino Uno, works at 3.3V and uses two I/O pins. The shield is equipped with a low-cost generic Bluetooth module that can only work as a slave device with limited functionalities. This module can communicate at 460 kbps up to 10 meters with Bluetooth master devices, such as a computer USB dongle, and does not support audio communication. Wireless communication between shields is not possible, but the software API compensates by being simple to use and understand.

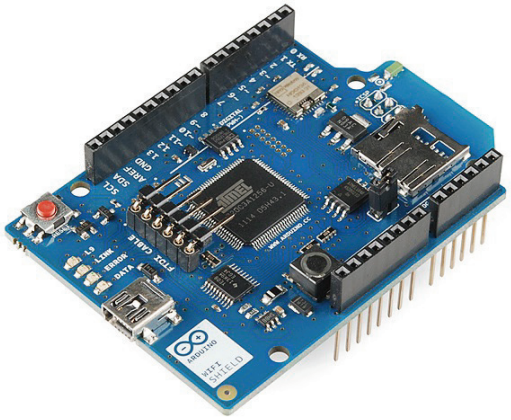
Figure 2.5 - Arduino wireless shields on the market.

(a) ITead Stackable Bluetooth Shield

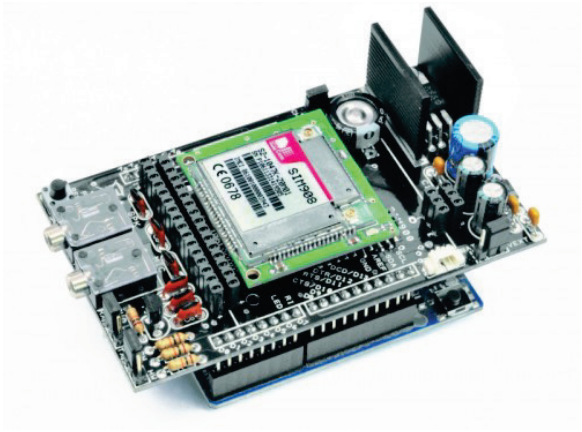


(b) Arduino XBee Shield

(c) Arduino WiFi Shield



(d) GSM/GPRS Shield



All Zigbee shields available in the market use standalone XBee wireless modules from Digi [15], as seen in Figure 2.5.(b). The functionalities added to the Arduino depend on the XBee model attached to the shield. Generally, the shields are sold with a normal XBee which allows data communication up to 120 meters at 250 kbps. More expensive versions come with an XBee-Pro, allowing the same data rates but up to 1.5km of distance. Xbee modules support all standard Zigbee features, but the communication API only allows to send data packets from point-to-point or from point-to-multipoint. For additional features, it is necessary to use AT commands. The Zigbee network is automatically generated by the modules.

The official Arduino Wi-Fi Shield [16], shown in Figure 2.5.(c), is one of the most complete wireless solution for Arduino. This shield contains an integrated Wi-Fi module and a 32-bit Atmega 32UC3 working as a co-processor for controlling. The 32UC3 firmware runs a TCP/IP stack and allows TCP and UDP connections over Wi-Fi by using the module. This shield is open-source and the API covers every aspect related to Wi-Fi, allowing encryption and supports client/server applications for example.

An Arduino shield for GSM/GPRS communications also exists [17] and is present in Figure 2.5.(d). A SIM900 [18] module from Micron Electronics is used for GSM and GPRS related operations.

The shield allows to establish TCP connections and to make and accept calls using GSM. For audio interface, the shield has on-board connectors for a headset. The API used is very complete, covering all functionalities supported by the module. The shield contains some of the features included in the Wi-Fi shield API, such as a TCP/IP stack.

Table 2.3 presents a summary of the characteristics and features of the shields described. It is possible to see that the cost of each shield depends greatly on technology, functionalities and range. The DECT shield has to be a low cost solution similar to the XBee Shield, has to have low power consumption, medium transmission range and an API which is able to provide an interface with most DECT functionalities when compared to the other shields.

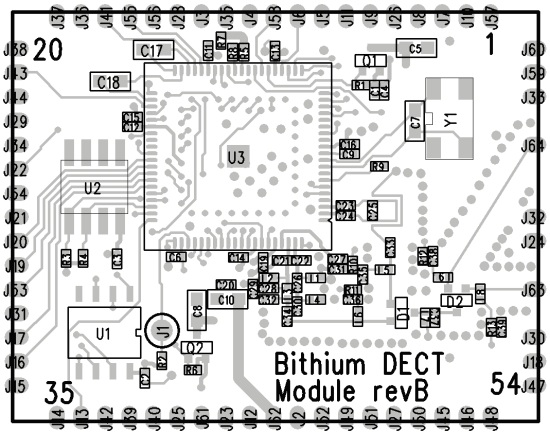
Table 2.3 – Wireless shields characteristics and features.

Shield	Cost (average)	Functionalities / API	Consumption	Range
Stackable Bluetooth Shield	15€	Limited	Low	Short
XBee Shield	35€	Limited	Depends[3]	Depends[3]
WiFi Shield	70€	Complete	High	Medium
GSM/GPRS Shield	60€[4]	Complete	High	Long

2.4 BITHIUM DECT MODULE

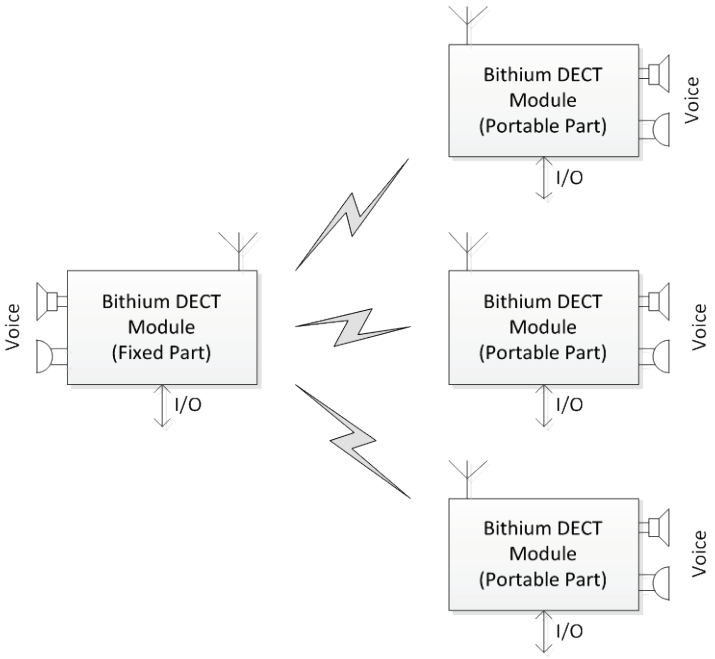
The Bithium DECT Module is illustrated in Figure 2.6 and was designed by Bithium. It is a 63 pin board with a size of 39.3 mm by 30.6 mm and is based on the SC14CVMDECT [19] module with several modifications. The module is capable of working as a DECT fixed part or as a portable part by changing the firmware. Typical operation between devices is illustrated in Figure 2.7. The module requires a 3.3V power supply, a few external bypass capacitors, two filters for voice communication and an external antenna.

Figure 2.6 - BDM layout and dimensions.



[3] Depends on XBee model.
[4] Additional costs for SIM card and carrier contract.

Figure 2.7 - BDM typical communication.



The core of the module is the SC14441A [20] operating at 1.8V. This integrated circuit (IC) contains an integrated 1.7 to 1.9 GHz CMOS radio transceiver for DECT communications, a programmable DSP for audio encoding and decoding, analog and audio interfaces with internal DACs and ADCs, a Class-D amplifier for 0.5W loudspeakers, a power management circuit and several standard connections, such as I/O ports, UART, SPI, I2C and PCM.

This IC also contains a 16-bit CR16Cplus general processor operating at 82.944 MHz that is used to control the components inside the chip. The processor runs a custom firmware that can be easily changed using an internal boot loader through the serial port. The firmware is stored in a 2 MB Flash Memory IC present in the module that is connected to the SC14441A thought a Quad SPI interface.

2.5 External Network Interface

The External Network Interface layer is an API developed by Bithium, which interacts directly with the DECT stack, allowing to perform several operations regarding DECT in a standardized way. This includes handling calls, data packets and the registration process for example.

The programming method employed by the API consists in an event base system, which has an idle loop, where non-critical and non-DECT related code is executed, and several user defined callback functions related to DECT mechanics. The callbacks are executed asynchronously when a DECT operation is executed or when there is a new status update for example.

The event based programming model, used in the ENI, has many advantages when working with user interfaces, such as buttons or LCDs, because certain elements only need to be updated or changed when a DECT event occurs. It is also very good because it encapsulates all DECT related components and makes them easy to access and process. However, it is difficult to create time critical applications for example.

This API works for fixed parts as well for portable parts, but both cases cannot coexist in the same firmware due to the way it is implemented. In order for the BDM to work as fixed part and portable part, a dual boot system can be created using two separated compiled binaries.

3 IMPLEMENTATION

3.1 REQUIREMENTS

The main goal of this project is to develop a shield that allows the Arduino to use the functionalities supported by the Bithium DECT Module. The shield needs to be competitive enough to compete in a market already lead by shields for other wireless technologies. It must be cheap, easy to use and support attractive functionalities for wireless applications. The DECT Shield is conceptualized after analyzing the DECT specifications, the BDM, the Arduino and the commercially available wireless shields in Chapter 2.

A list of requirements is presented:

- Use of Bithium DECT Modules for DECT operations.
- Ability to use the shield as a DECT fixed or portable part.
- Ability to perform DECT calls and data transfers between shields.
- Ability to reproduce and transmit an audio stream.
- Support for a headset and a loudspeaker.
- Ability to regulate speaker volume and microphone gain.
- Support for bidirectional communication between the Arduino and the shield.
- Easy to use Arduino API for shield interface.
- Support for asynchronous events.
- Shield dimensions similar to an Arduino Uno.

The final solution has to consist of an Arduino library, which can be used in sketches, a custom firmware for the BDM, which supports all features, and at least two fully assembled shields for testing.

3.2 DECT SHIELD

Some decisions and considerations are taken into account, using the requirements defined previously, in order to design the DECT Shield hardware and software. This section provides an overview of the components that constitute the shield, including some of the decisions involved in the design and planning procedure. A detailed description is provided in the next sections.

It was decided to place the BDM responsible for the shield control and the DECT communication handling, allowing the Arduino to execute other tasks in parallel. The BDM is significantly more powerful than the Arduino due to a superior CPU architecture, clock frequency and memory available.

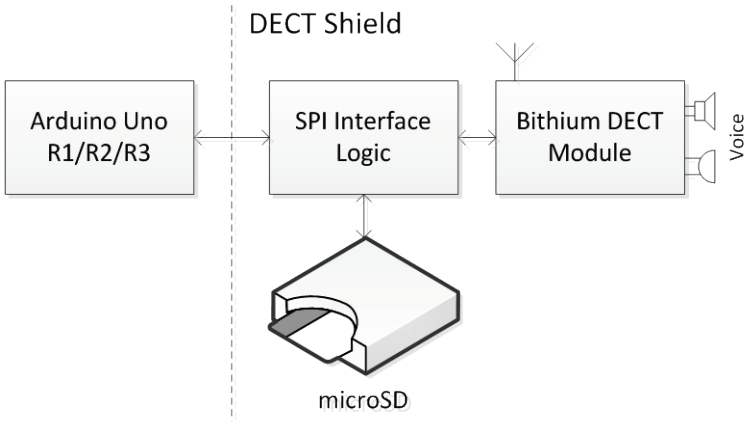
For that reason, this approach is efficient, reducing significantly the code size of the Arduino sketches and the number of I/O pins necessary to connect the shield to the Arduino.

To control the shield, a command based message protocol is specified and implemented. The Arduino is able to send commands to the BDM containing instructions and the BDM can interpret them (e.g. make call, send data). Each command has a specific operation associated and the protocol allows sending a payload related to the command. The BDM is able to send similar messages to the Arduino in order to respond to previews commands or to transfer received data payloads from a DECT link for example. The message protocol is fast and is implemented in the External Interface Layer (EIL) library, which has mechanisms to avoid and mitigate communication errors.

To enhance the shield functionalities, support for a microSD card was decided to be added. This allows to store files and increases the number of possible application in which the shield can be used. The BDM can open an audio file stream, from one of the files inside the card, and play it. Additional functionalities, such as audio streaming between shields or support for recording, can also be implemented. To improve playback performance, it was decided to allow the BDM to access the microSD card directly, without using the Arduino as an intermediary, but also to allow the Arduino to access the microSD card if required. A system to control the access to the card is implemented since more than one device will be able to use it.

Figure 3.1 presents an illustration of the DECT Shield architecture, including the connections between the Arduino, the BDM and the microSD, which are made using SPI [21]. The three devices are connected to the same bus as illustrated in the figure, but since they do not have same voltage logic levels, it is necessary to use a hardware interface circuitry.

Figure 3.1 - DECT Shield architecture.



SPI is selected because of high transference rates (8 Mbps, 50 Mbps [22] and 20 Mbps respectively) and native support by all three devices. The internal UART peripheral in the Arduino cannot be used, because it is already reserved for the serial port, and I2C [23] [24] does not have enough bandwidth to transfer audio data. Furthermore, I2C has a maximum clock frequency of only 400 kHz and the communication protocol imposes an overhead. The downside of SPI is requiring a

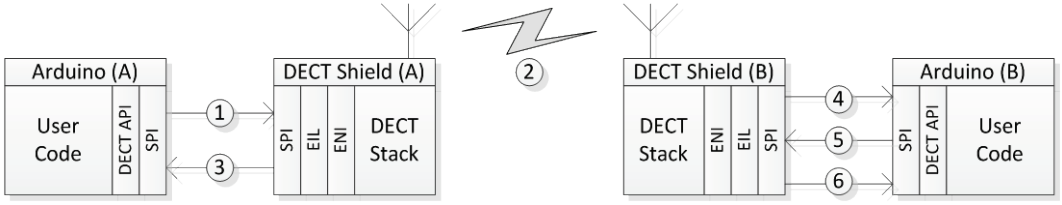
higher pin count and lacking error checking mechanisms when compared to other communication methods.

Arduino sketches have access to a library in order to interact with the shield, the DECT API library. This library provides a set of functions which are used to write Arduino applications using the shield. The DECT API encapsulates the EIL to control the shield using commands and supports two programming models: polling and events.

The BDM uses a proprietary DECT stack which is accessed using the External Network Interface. The EIL in also included in the BDM firmware and uses the ENI to control the DECT stack and act accordingly to the commands received from the Arduino.

The conceptualized mechanism for transmission of generic data between shields is represented in Figure 3.2. The figure also illustrates how the software libraries described interact with each other, providing an overview of the software components present in the Arduino and the BDM.

Figure 3.2 - Data communication between two DECT shields.



A sequence of operations consists of the following:

1. Arduino (A) sends a message, containing a command and a payload, requesting DECT data transmission.
2. The BDM on the transmitter shield (A) sends the payload through the air to the receiver shield (B).
3. If the payload was successfully sent, the transmitter shield (A) responds by acknowledging the Arduino, otherwise it responds indicating a failure.
4. An event is sent by the receiver shield (B) to inform its Arduino (B) that a new payload has arrived and is available for reading.
5. Arduino (B) at the receiver part acknowledges the event by sending a message to the shield (B), requesting the payload which was received.
6. The receiver shield (B) acknowledges the received command and transfers the payload to the Arduino (B).

3.3 HARDWARE

This section provides a detailed description of the hardware components and how they are used. A description of the shield printed circuit board design is also provided. The Production Dossier, including the circuit schematics, Bill of Materials (BOM) and layout Gerber files, is in Appendix B.

3.3.1 BITHIUM DECT MODULE INTEGRATION

The BDM provides several types of interfaces and internal peripherals as already stated (section 2.4). The DECT Shield uses the following:

- SPI for microSD card and Arduino communication (explained in section 3.3.2).
- UART for in-system programming (ISP) and debugging.
- General purpose I/O ports for shield control.
- Speaker and loudspeaker outputs for audio.
- Microphone input for audio capture.

The UART ISP connection allows to update the BDM firmware and provides a debugging interface without requiring a special programming device. Any UART interface cable operating at 1.8 V can be used for this purpose, such as a USB to UART converter.

A total of seven I/O ports are used. Four I/O ports control the SPI interface. One is used as a handshaking mechanism between the Arduino and the BDM, mitigating synchronization problems, and another is used to generate asynchronous events for the Arduino. Finally, an additional I/O port is used as a current source to power a status LED.

Speaker, loudspeaker and microphone interfaces are used to provide external connections for a headset and a low impedance loudspeaker. Each connection has an on-board audio filter to improve quality.

BDM output ports can be configured as push-pull or open-collector. Push-pull allows for faster transition rates but operate at 1.8V, which is not supported by the microSD and the Arduino, as both require input signals of at least 2.0 V and 3.0 V respectively. An external logic level translator can be used to boost the output voltage. Ports configured as open-collector can operate up to 3.5 V and require a pull-up resistor. Signal transition speed and port current consumption is affected by the resistance of the pull-up [25]. Ports in open-collector can be connected to the Arduino and the microSD since the output voltage is inside the detection range.

The maximum voltage of BDM input pins is 3.5 V. Therefore, the inputs can be connected directly to the microSD card outputs which operate at 3.3V. Arduino outputs operate at 5V and are reduced using logic translators and resistor dividers.

Transition times of signals required to control the SPI circuitry are not critical. Therefore, it was decided to use open-collector configuration. SPI interface ports are configured as push-pull, because data rates can become affected by the slower transition times. The SPI signals are converted using logic translators instead.

The pull-ups have a resistance of 4.7 kΩ. The average current consumption per pull-up, when the output is at low level, is 0.7 mA at 3.3V. Higher resistor values would reduce current consumption but increase transition times.

3.3.2 SPI INTERFACE

SPI communications in the shield are a major concern of this project. The Arduino, BDM and microSD work at different voltages and their SPI interfaces operate differently as presented in Table 3.1. In the Arduino, SPI data pins change direction based on the mode of operation, for example, the MOSI pin operates as output when the Arduino is master and as input when slave. The BDM and the microSD card pins do not change direction. These differences imply the SPI bus cannot be connected to all three devices directly.

Table 3.1 - SPI pins for Arduino, BDM and microSD card.

SPI pin	Arduino (Master)	Arduino (Slave)	BDM (Master)	BDM (Slave)	microSD
SPI Clock	SCK (output)	SCK (input)	SCK (output)	SCK (input)	SCK (input)
SPI Data Out	MOSI (output)	MISO (output)	DO (output)	DO (output)	DO (output)
SPI Data In	MISO (input)	MOSI (input)	DI (input)	DI (input)	DI (input)
SPI Select	N/A	CSS (input)	N/A	DSS (input)	CSS (input)
SPI Voltage	5V (3V+ inputs)	5V (3V+ inputs)	1.8V/3.3V	1.8V/3.3V	3.3V

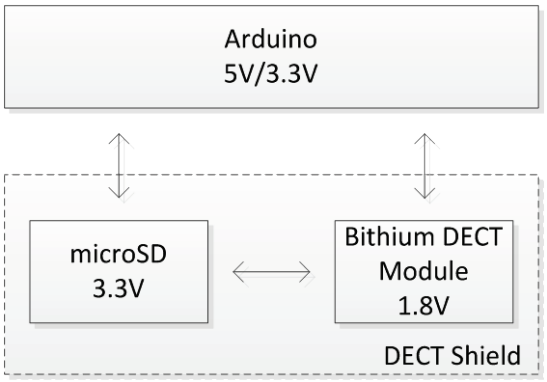
The BDM has only one SPI interface. Therefore, it is not possible to connect the BDM to the Arduino and to the microSD card using separated busses. Two different approaches are developed and tested to solve this problem.

Firstly, generic I/O ports are used to emulate a SPI interface connecting the BDM to the microSD card. This solution proved to be inefficient. The maximum data rate is 350 kbps and requires several CPU cycles to emulate the interface. A test showed the data rate is not enough to play an 8-bit / 8 kHz uncompressed audio file from the microSD card, producing gaps in the audio stream. The emulated SPI cannot be connected to the Arduino and operate as slave, because it does not have enough reaction time to latch incoming bits synchronously with the clock.

The selected solution consists in sharing a single SPI bus between the Arduino, BDM and microSD card. This method provides high data rates but requires external tri-state buffers and transceivers to route the necessary SPI signals to the Arduino or to the microSD card, forming a channel. Additional control signals are also required to operate the buffers.

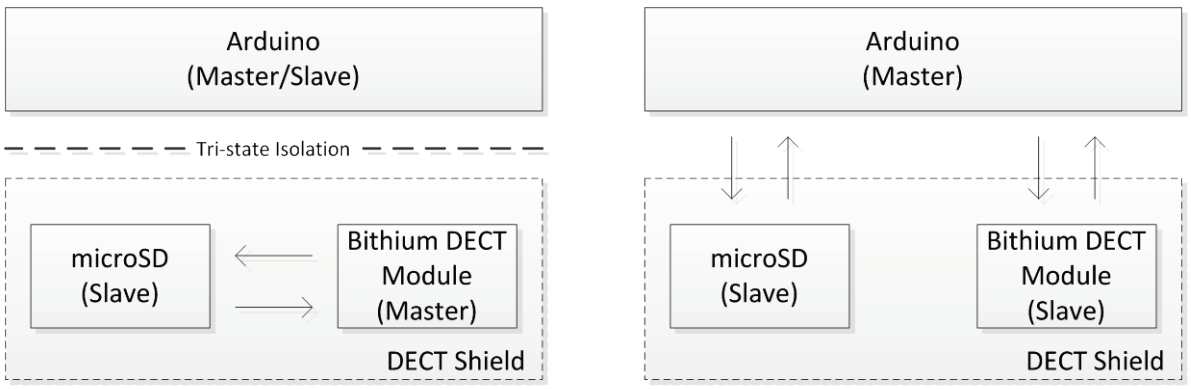
Figure 3.3 illustrates the possible SPI channels between the Arduino, BDM and microSD card. The Arduino always operates as master, the microSD card as slave and the BDM changes from slave when receiving data from the Arduino, to master when accessing the microSD card. Additionally, the channels connecting the Arduino to the shield have to be in tri-state when the BDM is accessing the microSD card, allowing the Arduino to communicate with other shields using SPI.

Figure 3.3 - SPI communication channels between the Arduino, BDM and microSD card.



Two distinct modes of operation can be identified, named Mode A and Mode B, because the BDM cannot operate as master and slave at the same time using a single SPI interface. Figure 3.4 illustrates the distinctions between the two modes.

Figure 3.4 - SPI communications channels in Mode A and Mode B.

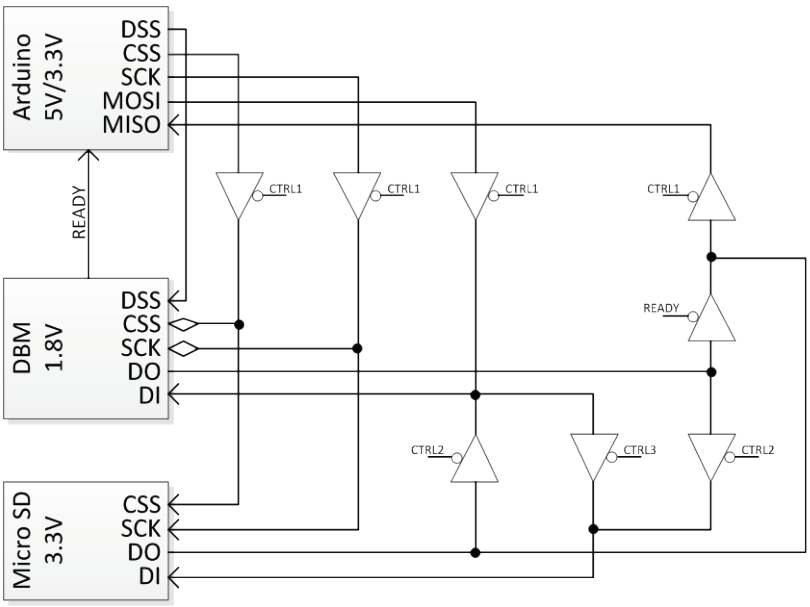


Mode A allows the BDM to read files from the microSD card and the Arduino to communicate with other shields. In this mode, the SPI bus connecting the shield to the Arduino is in tri-state, which allows communication with other shields without interference from the DECT Shield. The BDM operates as master in order to access the microSD card.

Mode B allows the Arduino to access the BDM and the microSD card. The Arduino operates as master and other shields cannot use the SPI bus. The BDM is in slave operation in order to receive data from the Arduino and cannot access the microSD card.

Figure 3.5 illustrates the schematics diagram of the circuit that shares the SPI bus between the Arduino, BDM and the microSD card. DSS and CSS are the SPI selection pins for the BDM and the microSD card, respectively.

Figure 3.5 - SPI tri-state buffer schematics diagram.



Shield tri-state isolation in Mode A is achieved using four buffers connected to Arduino SPI signals: SCK, MOSI, MISO and to CSS. The four remaining buffers define the active SPI channel. Control is performed by the BDM, reducing the number of necessary Arduino pins.

By default, the shield operates in Mode A. In order for the Arduino to communicate with the BDM, it has to assert the DSS pin. The BDM changes the shield operation to Mode B and asserts the READY signal, informing the Arduino to start transmitting. The READY signal is used as a synchronization mechanism to prevent the Arduino from sending commands before the BDM is ready.

Figure 3.6, Figure 3.7 and Figure 3.8 illustrate all three possible SPI channels in green. In Mode A, there is always a channel defined connecting the BDM to the microSD card. In Mode B, a channel is created based on the selection pins, CSS or DSS. Asserting CSS creates a channel connecting the Arduino to the microSD card and asserting DSS connects the Arduino to the BDM.

Figure 3.6 - SPI buffers in Mode A, BDM connected to microSD.

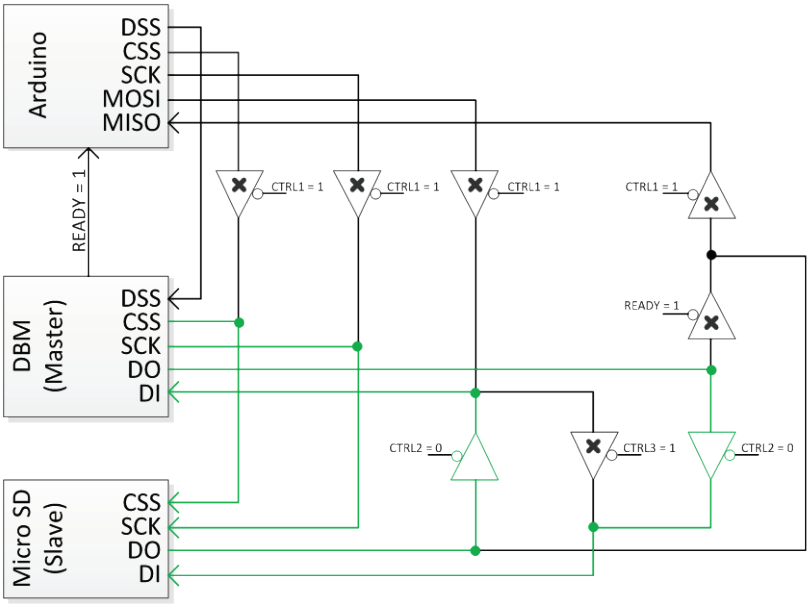


Figure 3.7 - SPI buffers in Mode B, Arduino connected to BDM.

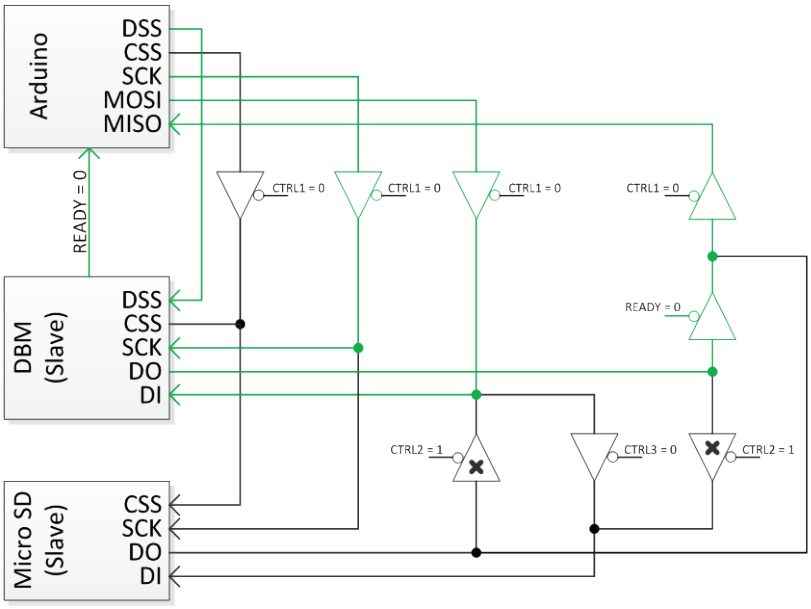
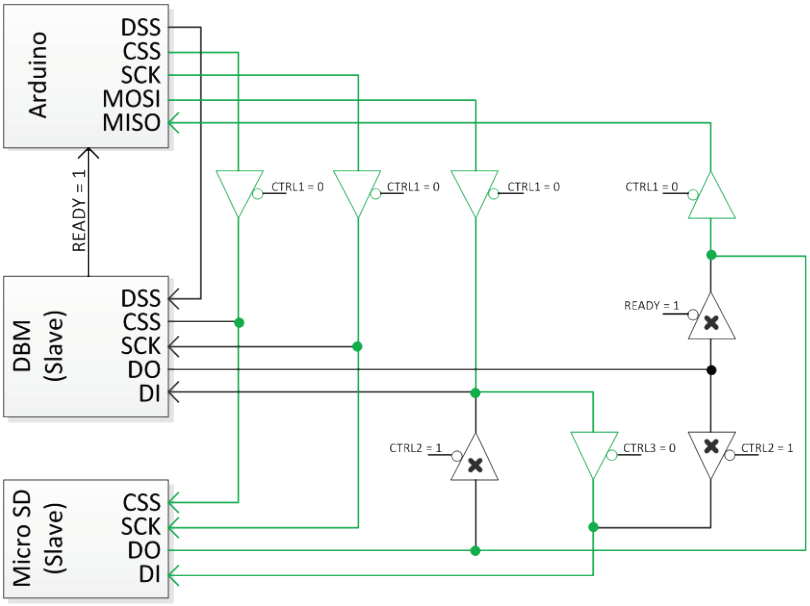


Figure 3.8 - SPI buffers in Mode A, Arduino connected to microSD.



In order to implement the SPI circuitry, the buffers need to be:

- Activate low - Buffers are disconnected when the shield is powered on.
- 5 V tolerant inputs - Arduino outputs can be directly connected to the buffers.
- Low propagation delay (< 5 ns) - Preserves SPI data rates.

The propagation delay is recommended to be lower than 10% of the minimum SPI period of the BDM, which is 50 ns. 5 ns does not affect the signal propagation significantly.

The buffers used in the shield [26] comply with the requirements and have a propagation delay of 3.9 ns when operating at 3.3 V. The pull-ups in the control lines force the activation signals to be high when the shield is powered.

The logic transceiver to boost the voltage level of the BDM SPI outputs has the following requirements:

- Bidirectional - For the SCK and CSS signal which operate as input and output.
- Low propagation delay (< 5 ns) - Preserves SPI data rates.
- 1.8 V to 3.3 V conversions.

The transceiver used in the shield [27] meets the requirements and has a maximum input-output propagation delay of 4.5 ns when operating from 1.8 V to 3.3 V.

Some tests were executed to boost the signals using a single MOSFET and two pull-ups [28] per signal instead of a transceiver. Although the results were acceptable, the necessary MOSFETs to work at 1.8 V were much expensive than using a single transceiver IC.

Appendix A contains the SPI timing analyses performed to determine the best SPI clock frequencies to be used by the Arduino, BDM and microSD card. The analysis showed that the Arduino

can use its maximum frequency of 8 MHz when communicating with the BDM and the microSD card. The BDM can use up to 14.7 MHz, when accessing the microSD card, due to delays imposed by the buffers and transceivers

3.3.3 AUDIO CONNECTIONS

The shield is designed to include two mono audio jacks: one to connect a speaker and another a microphone. A headset with separated plugs can be connected to the shield using these jacks.

Figure 3.9 presents the schematic diagram of the audio filter and the speaker connection. The audio is outputted from the BDM differential audio source connections, LSRP and LSRN, and allows to connect a speaker with an impedance of at least 28 Ω. The two 10 pF capacitors are used to filter high frequency noise mainly from the DAC conversion. The two 10 Ω resistors prevent damage to the BDM in case of short-circuit. The differential output has an impedance of 20 Ω that avoids any permanent damage.

Figure 3.9 - BDM to speaker connections.

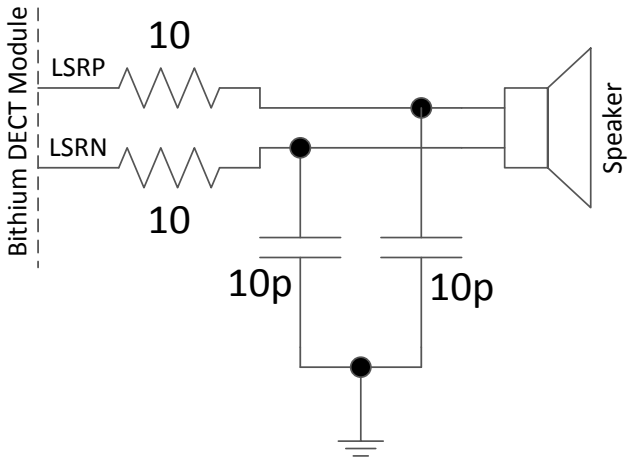
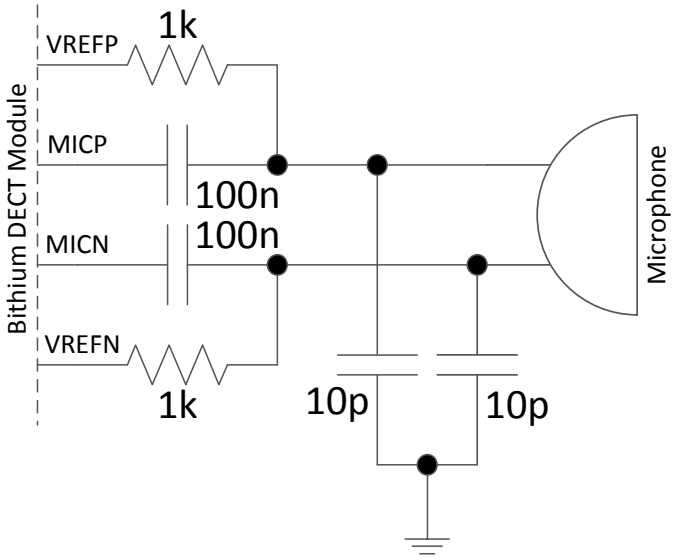


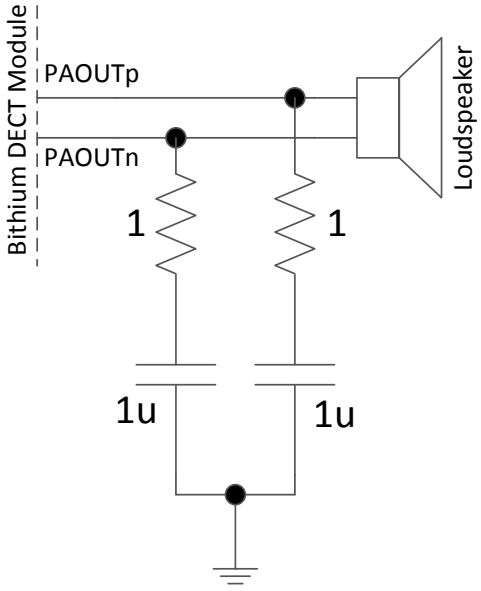
Figure 3.10 presents the schematic diagram of the microphone circuitry and connection. The BDM uses a variable voltage source to provide power on pins VREFp and VREFn, and a comparator to detect the microphone vibrations on pins MICp and MICn. The 10 pF capacitors filter high frequency vibrations, the 1 kΩ resistors bias the microphone and the two 100 nF capacitors are used to couple the signals before going to the comparator.

Figure 3.10 - Microphone to BDM connections.



The shield also includes a loudspeaker output through a 2-way terminal block. Figure 3.11 presents the loudspeaker circuitry and connection. The Class-D amplifier from the BDM is used and allows to connect a low-impedance speaker with at least 4 Ω. The 1 Ω resistors and the 1 μF capacitors are used to filter high frequency noise from the amplifier.

Figure 3.11 - BDM to loudspeaker connections.



3.3.4 Power Regulation and Distribution

The current necessary to power the components in the shield is estimated, based on the datasheets, to define the characteristics of the regulator to be used. Table 3.2 contains the average and peak currents of each component present in the shield.

Table 3.2 - Shield average and peak current consumption.

Component	Average current (mA)	Peak current (mA)
BDM	4.5	70
Loudspeaker	-	500
Logic ICs	4.7	7.2
microSD	0.1	60
Pull-ups	1.2	3.5
Total	**10.5**	**140.7 (+500)**

In a worst case scenario (Table 3.2), the shield consumes 140.7 mA and has 640.7 mA peaks.

Since the BDM works at 3.3 V, it can be powered directly from the Arduino 3.3 V on-board regulator. However, this regulator can only provide 50 mA of current which is not enough for certain situations (e.g. shield transmitting). Therefore, a 3.3 V regulator [29] is included in the shield to generate 3.3 V from the Arduino 5 V output pin. The regulator can output up to 800 mA.

The Arduino obtains 5 V from the USB connection or from a 5 V linear regulator, when being powered by an external source. The 5 V pin can output up to 500 mA of current that is enough to power the shield.

To improve power transmission and mitigate some noise [30], several bypass capacitors are used in the circuit near each component. 100 nF ceramic capacitors are adjacent to the power pins of each IC, microSD and BDM, and a 1 µF is near the BDM antenna circuitry power pin.

Additionally, two electrolytic bypass capacitors are used with the voltage regulator to improve stability and allow peak current surges during DECT transmission bursts or loudspeaker usage. One is connected to the regulator 5 V input pin and the other to the 3.3 V output. Both have a capacitance of 22 µF.

3.3.5 Arduino-Shield Interface

The DECT Shield uses six I/O ports and power connections from the Arduino. An additional optional pin is used to generate events. Table 3.3 presents each pin and its functionality.

Table 3.3 - Arduino pins for shield interface.

Pin	Name	Description
5V	-	5 V Power
GND	-	Ground
2	INT0	(Optional) Shield Events Signal
8	READY	BDM Ready Signal
9	DSS	BDM SPI Slave Selection
10	CSS	microSD SPI Slave Selection
11	MOSI	SPI Master-Out / Slave-In
12	MISO	SPI Master-In / Slave-Out
13	CLK	SPI Clock

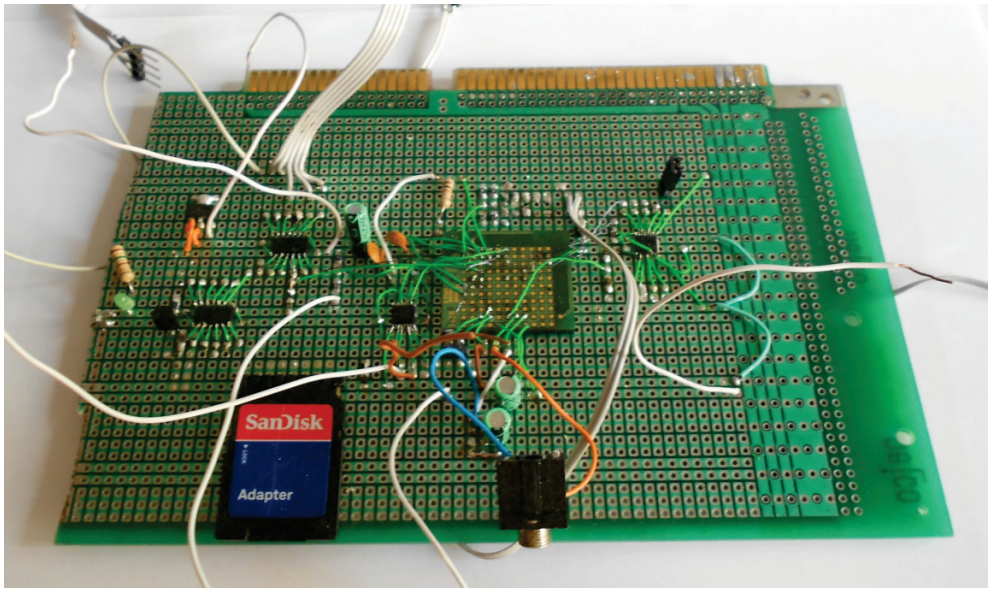
The SPI pins, MOSI, MISO and CLK, can be shared with other shields for SPI communications. Pins READY, DSS and CSS, cannot be shared because they are used to control the shield.

The shield can generate asynchronous events on the Arduino using the external interrupt pin, INT0. A buffer [31] is used in the shield to detach the signal from the Arduino when not in use, allowing other shields to use this pin when the DECT Shield is not configured to generate events.

3.3.6 Circuit Boards

The shield was first developed in a prototype board (Figure 3.12) using a perforated board. The prototype permitted to test all functionalities before investing in the final printed circuit board.

Figure 3.12 - Prototype board.



The first revision of the PCB is designed after executing the tests in the prototype board that allowed the verification of all functionalities. For the PCB design, a freeware version of Eagle CAD

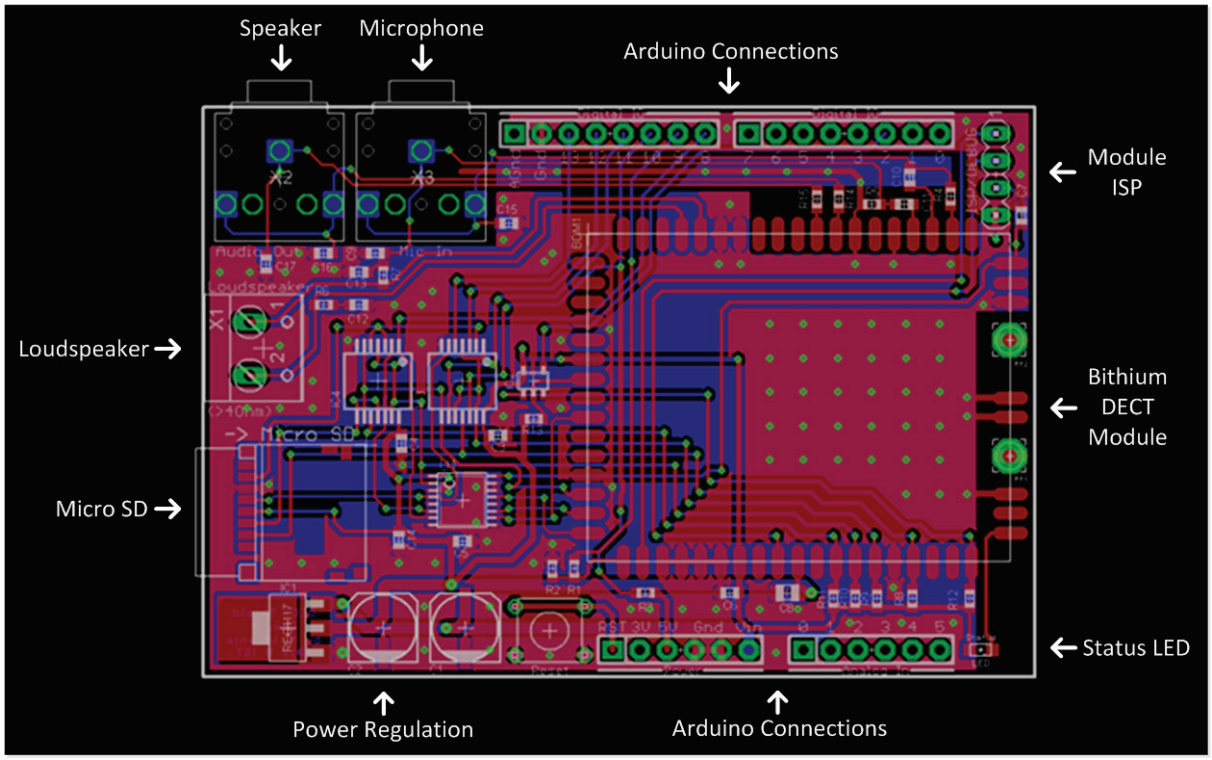
PCB [32] is used. The design software includes a schematic editor, a layout editor and a Design Rule Check (DRC) specification file from Eurocircuits [33].

Some considerations [34] and recommendations are taken into account when designing the shield:

- Use of 2 layers.
- DECT antenna needs to avoid pin headers and cooper areas.
- Distance between the antenna and the BDM needs to be as short as possible.
- Power traces from the BDM RF circuitry needs to be connected to the regulator without ramifications.
- Analog signals need to be separated from digital signals.
- Shield regulator needs to have additional cooper area to improve heat dissipation.
- SPI circuitry should be between the BDM, the microSD and the Arduino SPI pins.
- External connectors (speaker and microphone) should be placed close to each other.
- Power traces should have a width between 16 mil and 40 mil.
- Signal traces should have a width between 10 mil and 16 mil.
- Free circuit area needs to be left with cooper connected to Ground.

Figure 3.13 presents the final layout. The shield measures approximately 77.5 mm by 53.5 mm, being slightly larger than an Arduino Uno.

Figure 3.13 - DECT Shield layout (rev. A).



The BDM and the antenna are positioned on the right side of the shield in order to avoid the Arduino pins and jacks, allowing better reception and transmission. The antennas measure 5 cm and are soldered perpendicular to each other, forming a 90º angle.

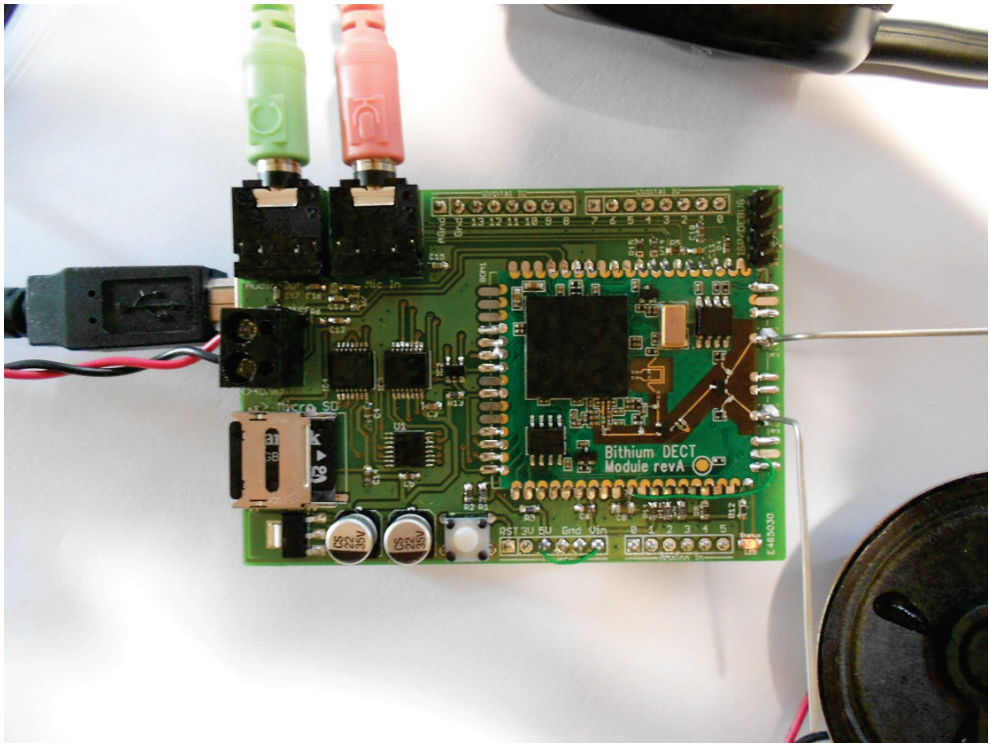
The speaker filter is near the audio out jack and the microphone filter is near the BDM, being as close as possible to the destination in order to filter interferences.

Bypass capacitors are located near each IC and the pull-up resistors are placed close to the BDM output pads. All ICs, resistors and capacitors used are SMD in order to conserve circuit area.

Free circuit areas left with cooper are connected to Ground in order to improve power distribution between the shield components.

The layout passed the DRC verification without errors and some PCBs were ordered from Eurocircuits. Figure 3.14 presents the DECT Shield fully assembled.

Figure 3.14 - DECT Shield assembled.

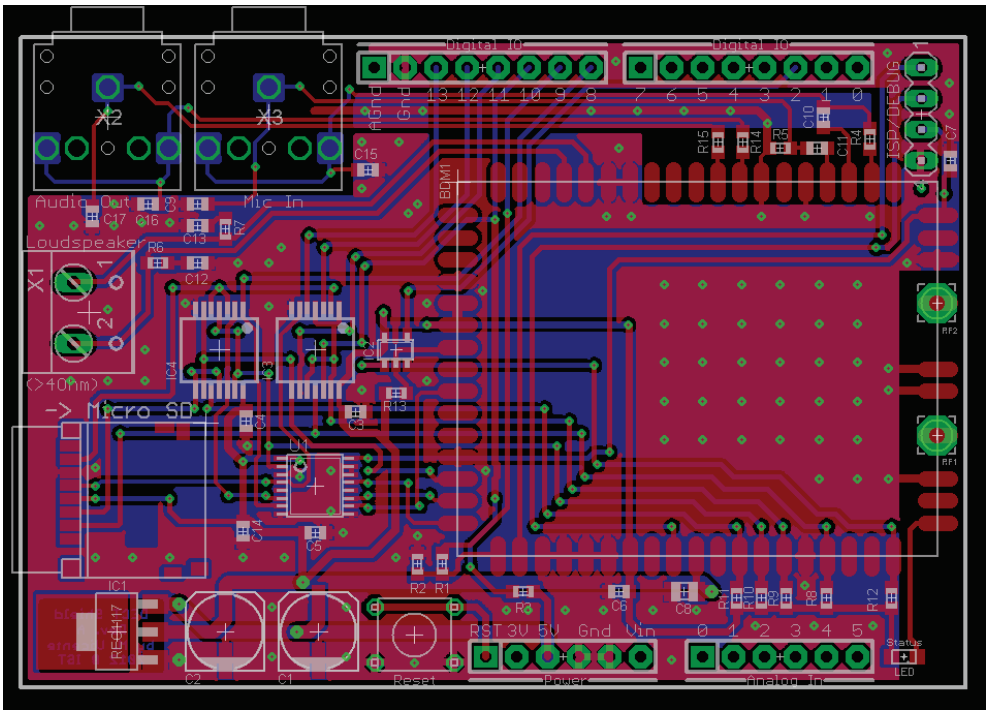


After some tests, there was a few problems detected, these are corrected in revision B:

- Changed the shield power pin from the Arduino V_{IN} to the +5 V pin. The V_{IN} can only be used if the Arduino is powered by an external source and the +5 V provides power regardless of the source (e.g. USB).
- Added a missing connection from the BDM VBAT pin to the 3.3 V power regulator.
- The size of some traces was reduced. For example, the connections to the loudspeaker

Figure 3.15 shows the final layout. More information is in the Production Dossier presented in Appendix B.

Figure 3.15 - DECT Shield layout (rev. B).



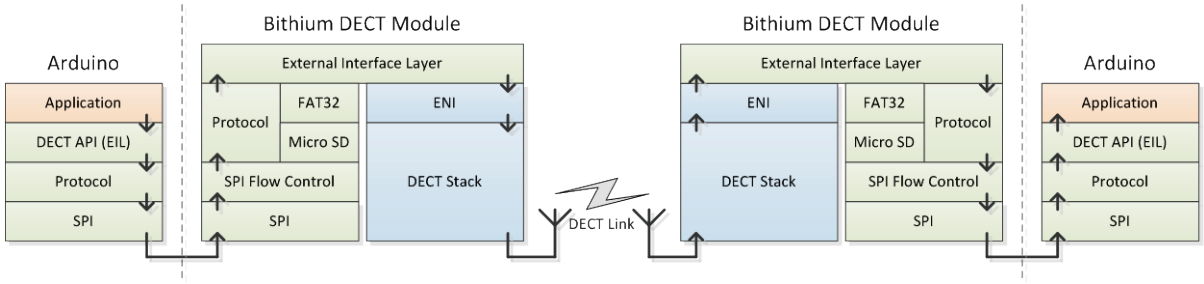
3.4 SOFTWARE

The next sections describe the software components and explain the implementation details. Detailed documentation of the libraries developed, including functions, structures and usage, is present in Appendix C and Appendix D.

3.4.1 OVERVIEW

Two major software components are developed in this project: a DECT API library to be used by the Arduino sketches and a BDM firmware. Both components consist of several abstraction layers, in form of libraries, that hide the implementation details and provide a set of functions for interfacing. An overview of all the layers and the interface between them is presented in Figure 3.16 for two DECT Shields. Layers in blue were developed by Bithium and green ones in this project. Arduino sketches are represented in red, Application.

Figure 3.16 - Software abstraction layers overview.



Low level communication between the Arduino and the BDM has similar abstraction layers, including the SPI layer, to send and receive messages containing the commands using buffers, and the Protocol layer, to generate/decode communication messages. The SPI Flow Control layer controls the SPI circuitry in the shield and is responsible to deliver payloads to the correct destination (Arduino or microSD).

The DECT API library provides a set of functions to be used in Arduino sketches, encapsulating the details of the EIL (section 2.5) which defines the commands to communicate with the shield. These functions use the Protocol layer to communicate with the shield.

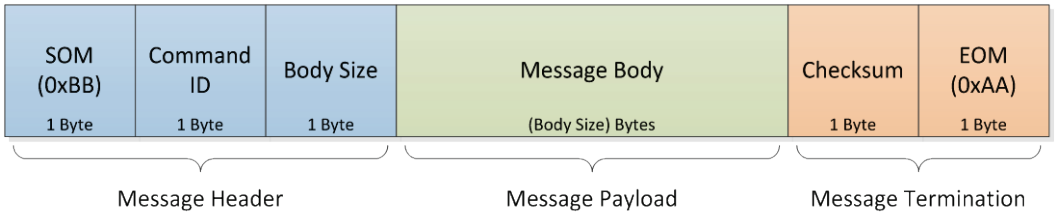
The messages received in the BDM are analyzed by the EIL, after been checked by the Protocol layer, which verifies their authenticity and executes the desired operation. This is the top-most layer because it interfaces all the lower level components that are used to communicate with the Arduino and to handle audio reproduction with the ENI.

DECT communication between shields uses the DECT stack composed by PHY, MAC, DLC, NWK and IWU OSI layers. The DECT stack is accessible through the ENI, a high-level API. Other DECT devices can communicate with the DECT Shield because the standard is the same.

3.4.2 MESSAGE COMMUNICATION PROTOCOL

The Protocol layer (Figure 3.16), in the Arduino and the BDM, generates and verifies messages used for communication between the DECT Shield and the Arduino. Each message contains a command identifier and content associated with a command. The layer is capable of detecting errors in received messages and is optimized for SPI usage. Figure 3.17 presents the format of the messages supported by the protocol.

Figure 3.17 - Protocol message.



Each message has the following elements:

- Message Header - Information about the command inside the message and the total size.
    - Start of Message (SOM) - Indicates a start of a new message.
    - Command ID - Identifier of the command present in the message.
    - Body Size - Size of the payload associated with the command (0 to 255).
- Message Body - Contents of the command inside the message (up to 255 bytes).
- Message Termination - Information to detect errors in the messages.
    - Checksum - Sum of all bytes in the header and in the payload.
    - End of Message - Indicates the termination of the message.

The messages have a minimum length of 5 bytes and a maximum of 260 bytes, depending on the command payload. An overhead of 4 bytes is present in each message (SOM, Body Size, Checksum and EOM).

The SOM and the EOM are used by the protocol to detect when the SPI library returns a start or an end of a message, instead of dummy or invalid data. The Protocol library records a message when a SOM is received and stops when an EOM is detected, when Body Size + 5 bytes are read or when a timeout is triggered. The checksum is used to verify if a received message contains errors.

A message is considered valid if the correct message length is read, starting from the SOM to the EOM, and if the checksum calculated locally is equal to the received. Valid messages are sent to the EIL in order to execute the associated command and invalid messages are discarded.

3.4.3 AUDIO PLAYBACK

The microSD layer and the FAT32 layer (Figure 3.16) are used to allow audio reproduction from files stored in the memory card. The microSD layer defines the interface between the microSD card using SPI and the FAT32 layer implements a FAT32 file system to open the contents stored in the card.

The microSD interface layer, based on the SanDisk Secure Digital Card specifications [35], includes the functions to initialize, read/write data block and to address the card through SPI.

The card initialization procedure contains a problem related to the BDM SPI interface, which the minimum clock frequency is of 741 kHz. Initialization must be executed at a maximum of 400 kHz, which is almost double of the minimum clock frequency supported by the BDM. A workaround is implemented to solve this problem by adding delays and repetitive attempts until the card is successfully initialized. After initialization, which can take up to 3 seconds, the clock frequency is changed to 10 MHz, increasing the data rate considerably.

The FAT32 layer is a generic FAT32 file system emulator, which in this case is interfaced with microSD cards. The implementation is based on the Microsoft FAT32 specification [36] and supports all FAT32 features, including long names and long directory entries. The developed function set for the FAT32 layer allows navigation through directories, file inspection, file creation and file removal, as well as opening files from a given path. Files are read/write in 512 byte blocks.

A file system facilitates copying files from a computer to a card without requiring special software and allows having multiple files organized by folders. The most used file systems for flash drivers and memory cards are FAT32 and NTFS. Both have advantages and disadvantages [37], but the FAT32 is selected because it is simpler, has lower overhead and is already supported by the Arduino API.

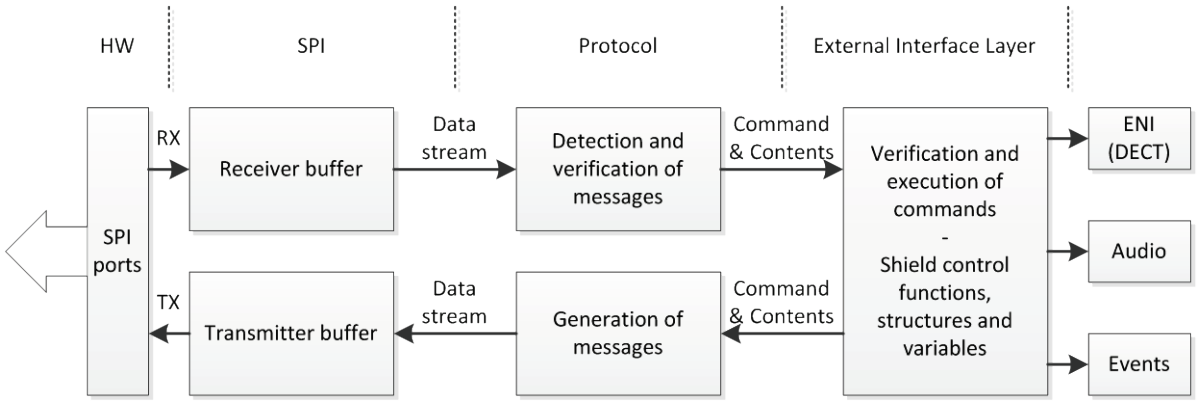
WAV format [38] was chosen for audio files, because it can contain uncompressed data in PCM or μ-law, which can be sent directly to the BDM audio processor without software conversions. A WAV interpreter, that analyses the files and determines the play time, the sample rate and the bytes per sample, is implemented. Current firmware supports only WAV files in 8-bit μ-law format at 8 kHz.

3.4.4 EXTERNAL INTERFACE LAYER

The External Interface Layer is responsible for verifying, execute and respond to commands sent by the Arduino. The EIL contains and manages the components that control the DECT Shield, including the ENI (section 2.5), the Protocol layer, audio playback and events. Detailed information about the commands defined in the EIL is present in Appendix C.

Figure 3.18 illustrates how the EIL is integrated with the remaining software components. Messages received from the Arduino are separated in the Protocol layer and extracted commands are send to the EIL. The EIL executes the commands and acknowledges to the Arduino by sending responses.

Figure 3.18 - Interface between abstraction layers.



The EIL contains several structures to hold shield information which the Arduino can access, and in some cases alter, using commands. The EIL has the following structures:

- A configuration structure that contains the shield current settings, regarding speaker volume, microphone gain, events, etc. Default configurations are applied when the shield is powered or after a reset.
- A status structure that holds status information, regarding the number of data packets available, the number of pending events, audio reproduction, etc.
- A FIFO structure to temporarily save received data packets from other shields. The oldest data packet is sent to the Arduino, when requested, and is removed from the structure. Incoming data packets are discarded when the FIFO is full.
- A FIFO structure to hold pending events. Triggered events are saved in this structure and the Arduino can request the oldest pending event from the top of the FIFO. Events are removed when requested and discarded if the structure is full.
- A variable array to hold a sequence of WAV files that will be played. The array size can be changed and files can be queue into this structure using the Arduino. Issuing a play command plays the sequence present in the structure.

The idle loop in the ENI (section 2.5) is used by the EIL to wait for commands and to read audio data from the microSD card during playback. Asynchronous callbacks from the ENI are triggered during DECT events and can add new elements to the structures, for example when a new data packet is received. Non-DECT commands, which do not use structures changed by the callbacks, are executed in the idle loop (e.g. playback commands, speaker volume). The remaining commands are queued and executed in a callback that is triggered every 40 ms, ensuring code atomicity.

The EIL also contains mechanisms to avoid audio stalls, or gaps, during playback. During a file reproduction, the Arduino can continue to communicate with the shield using commands, but the response time might be slower. The EIL only allows the Arduino to use the SPI bus to transmit a

command when an audio block is read from the microSD, ensuring that the Arduino has enough time to send and receive a message, before a new audio block is required.

For example, in a standard 8-bit μ-law / 8 kHz audio file being read at 10 Mbps, it is necessary to read a new audio block of 512 bytes every 64 ms, which takes approximately 4.1 ms. The Arduino can use the remaining 59.9 ms to send a command as soon as the block finishes transferring. FAT32 overhead is not taken into account in the example.

3.4.5 DECT API LIBRARY

The DECT API (Figure 3.16) is a high level library that allows the Arduino to use the DECT Shield in sketches. The library encapsulates the EIL and uses the command set with the Protocol layer to communicate with the shield. More detailed information about the API functions is in Appendix D.

The library contains the functions necessary to retrieve status information, execute operations and control the behavior of the shield. There are functions to use the DECT components, including registration and data/voice communication, to play files from the microSD card and to configure the external connections.

The components bellowing to the API (e.g. variables, functions, interrupts) are defined in a class included in the `DECT.h` header file. The class needs to be instantiated globally and is used to reference the DECT Shield. Configurations to the Arduino are applied when the class constructor is executed and events can be enabled during the declaration. Figure 3.19 exemplifies how the header and the class are declared in a sketch.

Figure 3.19 - DECT API class declaration.

```
// DECT Library
#include <DECT.h>

// DECT Class instantiation
DECT dect(false); // No events
```

Each function uses the Protocol layer to send a command to the shield and to wait for an answer. Error detecting mechanisms with timeouts are implemented, preventing the library from locking. All functions are executed atomically, when events are enabled, by disabling the interrupts temporarily.

The API provides two methods to control the shield: polling and events. The first method, polling, uses DECT functions to constantly get the module status and to act based upon the information retrieved. The second method generates asynchronous events, reporting a status change in the shield. Callbacks, defined in sketches, can be associated to one event and are executed when triggered.

3.5 APPLICATIONS

Table 3.4 presents the sketches developed to test the DECT Shield functionalities. The goal was to create a set of examples that cover all developed features present in the shield. For instance, the Terminal sketch permits to execute and analyze the shield functionalities using a bash terminal through the serial port. The Text-to-speech sketch exemplifies audio playback from the microSD card, and the Telephone demonstrates how to create a rudimentary voice telephone between shields.

Table 3.4 - Developed sketches.

Sketch Name	Functionalities tested				
	Events	Voice	Data	Playback	Audio
Terminal	Partial	Yes	Yes	Yes	Yes
Text-to-speech	No	No	No	Yes	Yes
Telephone	Yes	Yes	Yes	Yes	Yes

A computer is used in all applications to power the Arduino and the shield. Additionally, the computer is used as a serial terminal, printing received data to the monitor, and relaying keyboard inputs to the Arduino. The Arduino communicates with the computer using the serial port (UART) emulated by USB.

Standalone applications, without requiring a computer, can also be create. For example, wireless speakers that are powered by an outlet and receive music wirelessly from another shield.

3.5.1 TERMINAL

The Terminal sketch is a simple Linux bash-like terminal that permits to control and analyze the status of the DECT Shield. It allows to test and understand how the functionalities supported by the shield work, before creating an application to accomplish a certain task.

Figure 3.20 presents a screenshot of PuTTY, a multiplatform terminal, running on a PC that is connected to an Arduino running the Terminal sketch. The sketch receives and executes the commands through the serial port.

Figure 3.20 - Terminal running on Arduino using PuTTY as interface.

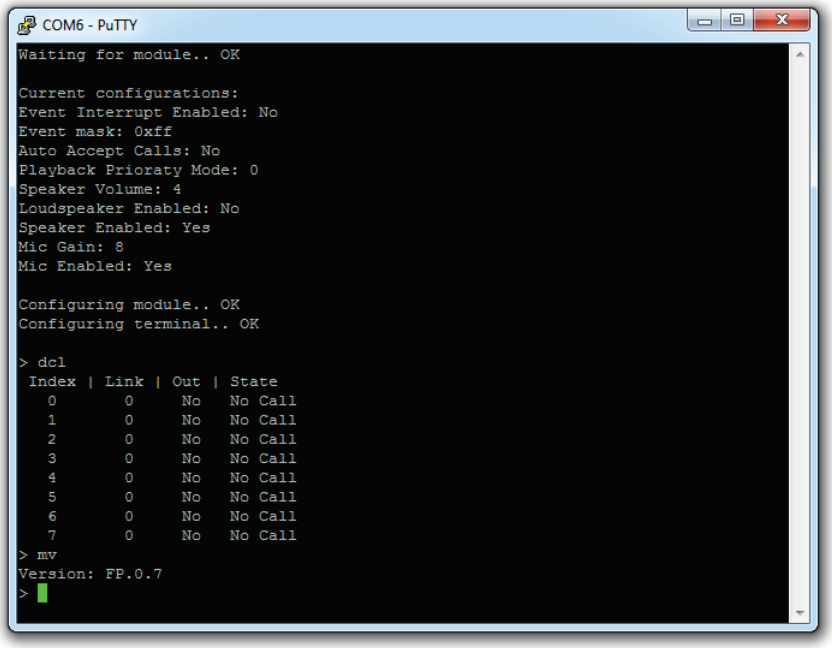


Table 3.5 lists of all commands supported by the application. The commands are separated in five groups and each corresponds to one specific set of functionalities supported by the shield. These are the following:

- Module - Control, configure and retrieve status information from the shield.
- DECT - Handle registration, data and voice communication between shields.
- Playback - Control and configure audio file reproduction from the microSD card.
- Audio - Configure the speaker, loudspeaker and the microphone connections.
- Memory Card - Control the microSD card access privileges.

There are some commands, such as *dpc* for the pin code and *dreg* for the registration procedure, which can be used to different purposes based on the arguments. For instance, if *dreg* is called without arguments then the current pin is returned, otherwise it will be change

Table 3.5 - List of commands supported by the Terminal sketch.

Command			**Arguments**	**Returns**	**Description**
Group	**Name**	**Terminal**			
Module	Version	**mv**	-	Version	Gets the current version
	Firmware	**mf**	PP/FP	-	Changes the firmware type
	Reset	**mr**	-	-	Resets the module
	Defaults	**md**	-	-	Reset with the default settings
	Status	**ms**	-	Status	Requests all status information
	Configure	**mc**	Configs	OK/Fail	Configure the module
	Event	**me**	-	Event	Request last pending event

Table 4.2 (continuation) - List of commands supported by the Terminal sketch.

DECT	Info	**di**	-	RFPI/IPUI	Return RFPI (FP) or IPUI (PP)
	Pin Code	**dpc**	Pin // -	Current Pin	Set or get the pincode
	List Modules	**dl**	-	Module List	Get the list of registered modules
	Send Data	**ds**	Pin	Ok/Fail	Send data to another module
	Receive Data	**dr**	-	ID, Data	Receive one data packet
	Call	**dc**	ID, Data	Ok/Fail	Make, accept, refuse or terminate a call
	Call List	**dcl**	-	Call List	Get call information list
	Register	**dreg**	On/Off // ID, Action	Ok/Fail	Enable registration for FP or register PP
	Unregister	**dureg**	-	Ok/Fail	Unregister from current module
	Scan	**dscan**	On/Off	Ok/Fail	Enable or disable scan on PP
	Scan List	**dsl**	-	Ok/Fail	Get the list of scanned modules
	Status	**dst**	-	DECT Status	Get the DECT status
Playback	Play	**pbp**	-	-	Plays the opened file or queued sequence
	Stop	**pbs**	-	-	Stops playing
	Volume	**pbv**	0-15	-	Changes the volume (0 - 15)
	Playback Status	**pbst**	-	Playback Status	Gets the playback status
	Open File	**pbo**	File path	OK/Fail	Opens a .wav file from SD Card
	Queue Size	**pbqs**	Size	OK/Fail	Changes the playback queue size
	Queue File	**pbqf**	File path	OK/Fail	Queue a file to be played
Audio	Speaker	**as**	On/Off	-	Enable or disable the speaker
	Loudspeaker	**als**	On/Off	-	Enable or disable the loudspeaker
	Speaker Volume	**av**	0-7	-	Change the speaker volume (2dB to -12dB in steps of -2dB)
	Microphone	**am**	On/Off	-	Enable or disable the microphone
	Microphone Gain	**ag**	0-15	-	Change the microphone gain (0dB to +30dB in steps of 2dB)
	Configurations	**acfg**	-	Audio Configs	Gets the audio current configurations
Memory Card	Access	**sdacc**	-	-	Allows the Arduino to access the microSD card
	Release	**sdres**	-	-	Releases the microSD card from the Arduino

3.5.2 TEXT-TO-SPEECH

This application example mimics Text-to-speech in the Arduino by using the DECT Shield to play audio files from the memory card, exemplifying file reproduction and shield configuration.

The Text-to-speech sketch receives a sentence from the serial port and identifies the different words. For each word, an audio file is played by the shield, if it exists. Audio files associated to words

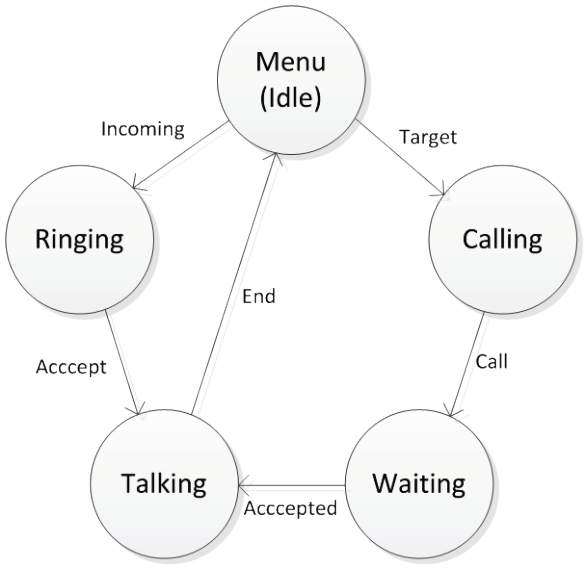
are stored in a microSD card and are organized in two level folders to reduce search time. For example, the audio file for the “hello” word has the following path “\h\e\llo.wav”. The Arduino converts each word into a file path and commands the shield to open and play the target file.

The audio files used for testing purposes were obtained from the AT&T Natural Voices [39] website, which allows inputting text and downloading the corresponding audio file. Each file downloaded is in PCM 16-bit / 16 kHz WAV format and was converted to μ-Law 8-bit / 8 kHz WAV format, which is compatible with the shield.

3.5.3 TELEPHONE

Telephone is a test sketch that emulates a rudimentary telephone between two DECT Shields, one working as portable part and another as a fixed part. The sketch permits to test events, voice communication, data transfer and shield configuration. The application implements a contact list, ringing, conversation and hang-up. Interaction with the sketch is performed through the serial port. Figure 3.21 illustrates the sketch state diagram.

Figure 3.21 - Telephone sketch state diagram.



The sketch uses exclusively events and the states change when a new event is triggered (e.g. keyboard input, call accepted). The applicant can be on one of the following states:

- Menu - Arduino is idle and the user can receive or make calls.
- Calling - Waiting for the user to select a target Arduino from the contact list.
- Waiting - Waiting for the target Arduino to accept or refuse the call.
- Talking - Call is undergoing, waiting for the user or the target to hang-up.

- Ringing - Incoming call detected, user needs to accept or refuse the call.

Data transferences are used when one telephone wants to know the name of another. Names are used to identify other shields running Telephone when the application switches to Calling state, showing the number and names of known parts.

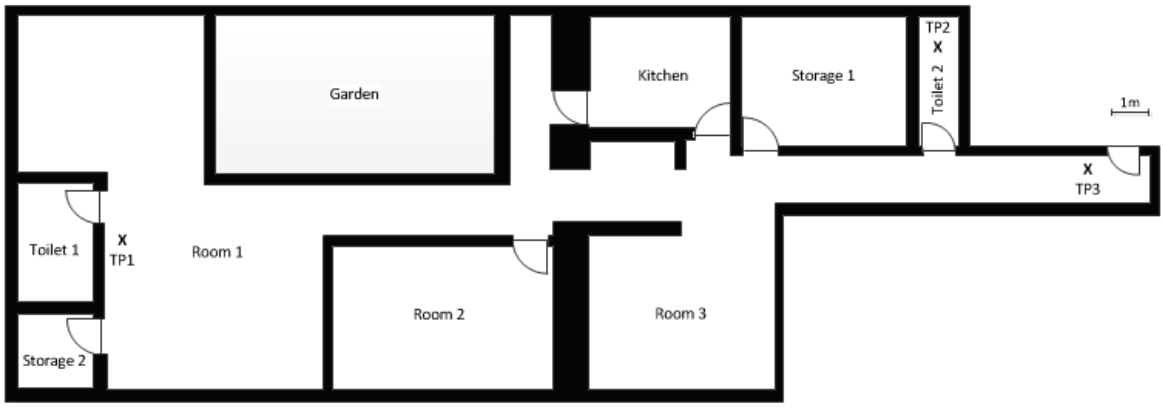
4 SHIELD EVALUATION

To verify the shield correct operation, some tests were executed using the three applications developed (section 3.5) and additional operation specific sketches targeted for debugging. It is possible to test the shield performance regarding range, transmission rates, responsiveness and consumption. Some comparisons with existing shields are also performed.

Transmission Range

The range tests were executed in-doors and out-doors. In-door tests were performed in the Bithium office, depicted in Figure 4.1, and the out-door tests in the countryside. The transmission range depends on the BDM performance, antenna and test conditions. The BDMs used in the shields were an early revision and contain a problem related to the antenna diversity. Therefore, the tests might be influenced and show lower transmission ranges than the final version.

Figure 4.1 - Bithium office Blueprint



The tests realized consisted in establishing several voice calls, using the Telephone sketch, and detect when the audio communication begins to show signs of interference or when one shield is unable to detect the other.

Tests in the Bithium office consisted in moving one shield to several locations inside the office and have the other shield stationary in TP1. The tests showed the shields are able to perform voice calls without interference at least 30 meters (TP3), with no obstacles, and up to 25 meters (TP2), with obstacles. The office dimensions limited the line of sight tests up to 30 meters in crowded DECT environments. Therefore, longer distances might be possible.

The only location in the office where the audio communication began showing signs of interference was in TP2. This room is covered with tiles and other construction materials that attenuate RF considerably, so it was expected to have interference in this location.

An out-door test in the countryside, a non-crowded DECT environment, was also executed in order to measure the maximum possible range the shields can achieve without obstacles. The test was executed by having one shield stationary and the other moving in a straight line with no obstacles in between. The test showed that the communication channel starts showing signs of interference at approximately 60 meters, which can be considered the maximum distance without interference in a best case scenario.

When comparing the communication range with similar shields such as the Bluetooth Shield and the XBee Shield, which guaranty communications up to 10 meters and 120 meters respectively, the measured values are within expectations even when the BDM contains an antenna problem and the DECT standard ensures reliable communication up to 100 meters.

Data Transmission Rates

Transmission rates were tested using sequences of data packets between two shields. The theoretical maximum data rates is of 1.2 kbps, since data communication is achieved using DECT control messages instead of dedicated data channels. Using control messages to transfer data between shields allows to have an audio channel established and at the same time transmit data in parallel.

The maximum transmission rates measured during testing were approximately 1 kbps. This value is close to the maximum supported rate and is better than expected. When compared to the Bluetooth Shield, which supports rates up to 460.8 kbps, and the XBee Shield, supporting rates up to 250 kbps, the DECT Shield comes as an inferior solution, but with native voice support. The data rates can be improved by adding support for dedicated data channels in a future firmware.

DECT Communication Handling

Several tests were performed to verify if the shields were handling DECT voice communication calls correctly. The tests consisted in establishing a voice call and then terminate or refuse by ending the call normally, turning off one of the shields or get out of range. The results showed some inconsistencies, which were corrected, when a call was lost.

Data communications did not show any errors, since the shield only recognizes an incoming package when it is received without errors.

SPI Communications

Tests were executed to detect errors in the SPI channel. Communication between the Arduino and the BDM is performed at 1 MHz and at 10 MHz between the microSD card and the BDM, respectively. Tests showed that the shield firmware controls the SPI circuitry correctly and no software lockouts were detected during stress tests.

A stress test was performed by sending several commands in sequence to the shield when reproducing a file. This test showed that the Arduino received all commands correctly and no audio gaps were detected at these frequencies.

Additionally, a test was created to generate invalid protocol messages in order to analyze the shield and Arduino behavior. The test showed the EIL detects the invalid messages and discards them, as it was programmed to do.

The Text-to-speech sketch was executed using a microSD card of 1 GB containing 109 577 different audio files spread across 702 folders, occupying a total of 951 MB, to analyze the SPI performance between the BDM and the microSD. The FAT32 library handled the card correctly but searching a file in a folder with more than 3000 files proved to be slow, taking approximately 1 to 2 seconds.

Shield Responsiveness

Communication between the Arduino and the shield is not immediate. A test using the oscilloscope was executed to determine the shield maximum delay time to start executing a command after the Arduino sends a message. The test showed the BDM can take up to 50 ms to start analyzing a command.

The measured response times were expected, since there is a limitation imposed the ENI. The delays consist of the time it takes for the Arduino to transfer a message, the time necessary for the BDM to detect a new message and to check for errors. The firmware in the BDM requires received messages in the idle loop to wait a maximum of 40 ms for a callback in the ENI to be triggered, reducing the responsiveness significantly.

The time it takes for the shield to respond to the Arduino after executing a command varies greatly from command execution time. Measurements show that commands that access the microSD card are the slowest and can take up to 3 seconds of execution. Commands that control audio are executed in less than 1 ms since they do not require access to the ENI. DECT commands can take up to 125 ms for data and voice communication and up to 5 seconds for registration operations. Finally, configuration commands generally take up to 50 ms. The times presented are after the BDM starts executing the command.

Programming Space

The DECT API library for the Arduino requires approximately 4 kB of programming space and 12 bytes of RAM. The API occupies approximately 12% of the programming space available in an Arduino Uno, leaving 28 kB free for user code or other libraries.

The combined firmware for the BDM (PP and FP) occupies approximately 220 kB and consumes half the RAM available to hold data structures.

Shield Consumption

Table 4.1 presents the average measured current consumption of the shield using an ampere-meter. The measurements are within expected values (Table 3.2) and no irregularities were detected.

Table 4.1 - Measured average shield current consumption with a microSD card.

Mode of operation	Portable part (mA)	Fixed part (mA)
Idle	9.5	21.0
Voice communication (w/o loudspeaker)	24.0	33.5
Voice communication (w loudspeaker)	55.5	64.0
Data transmission	19.5	31.0
File reproduction (w/o loudspeaker)	N/A	24.5

The shield consumes less when working as a portable part, because the DSP and the DECT circuitry inside the BDM are not fully activated until the creation of an audio channel or until a DECT transmission is executed. Measurements show the on-board regulator dissipates approximately between 15 mW and 96 mW, that is within expected values.

When comparing the transmission consumption with the Bluetooth Shield, which consumes approximately 20 mA, and with the XBee, which consumes 35 mA, it is possible to conclude that the DECT Shield consumes similar amounts of current, when working as a portable or as a base.

Manufacturing Costs

Table 4.2 presents the price estimation from the BOM of the DECT Shield, Bluetooth Shield and XBee Shield for comparison. The average prices were obtained from Mouser website at high quantities: price of 5000 units for passive components, 1000 units for semiconductors and 1000 units for connectors. The prices for the printed circuit boards were obtained at quantities of 300 from Iteadstudio. Calculating the approximate shield costs from the BOMs allows to do a fair comparison without using market value prices.

Table 4.2 - Shield cost estimation from BOM.

Component family	DECT Shield	Bluetooth Shield	XBee Shield
Passive components (e.g. resistors, capacitors)	0,28 €	0,10 €	0,31 €
Integrated circuits	1,32 €	0,00 €	1,21 €
Module (DECT/Bluetooth/XBee)	15,00 €	6,91 €	17,00 €
Connectors (e.g. audio jacks, header)	4,46 €	1,95 €	2,91 €
Printed Circuit Board	1,00 €	0,70 €	1,00 €
Total average price	**22,06 €**	**9,66 €**	**22,43 €**

The results present in the table show that the DECT Shield has a similar price to the XBee Shield but higher than the Bluetooth Shield, which costs approximately half. Although the prices are similar, the DECT Shield contains a microSD slot and audio connections, which are not present in the other

two shields. The average price of 22 € for the DECT Shield is within expectations considering the functionalities provided by the shield.

The microSD card slot and some of the logic associated with the interface does not require to be populated, which can reduce the cost of the shield by approximately 1,50 €, resulting in a price of 20,56 €.

5 CONCLUSIONS

The main objective of this thesis was to specify, design, manufacture and assemble a DECT Shield using a Bithium DECT Module, and to develop an easy-to-use library for Arduino. Sketches are written using the library, commanding the shield to register, handle calls and transfer data to other compatible DECT devices.

The shields are double layered printed circuit boards and their components are common, being easily acquirable. Therefore, the shield can be considered an undemanding “Do It Yourself” project that can be manufactured and assembled by hobbyists and independent developers.

The shield hardware, including the schematic, the layout, the BOM and the Arduino DECT API are open-source. The source code of the BDM firmware uses proprietary libraries and could not be distributed. However, the BDM can be updated using the shield ISP port and a binary file containing the firmware. All resources were made available online in the Arduino forums and positive responses were received.

For audio communications the shields support a headset (microphone and speaker) through two standard 3.5mm jacks and a low impedance loudspeaker. There is also an on-board microSD card slot which the Arduino and the BDM can access to store files. The shield can play audio files stored in the card using the speaker or/and the loudspeaker.

There were some difficulties when attempting to integrate the ENI with the Protocol and the microSD layers, where no user interfaces are used nor callbacks required. Both layers are time critical and hard to adapt to an event based system, which resulted in a mild impact in performance.

All functionalities supported by the software were tested in crowded and non-crowded DECT environments using the three test applications developed for this purpose. All features present in the shields are covered by these applications and work as expected, allowing to play files, register DECT devices and to transfer data and audio wirelessly.

The shield cost and supported functionalities are within expectations, being able to compete with existing short-range range shields, using Bluetooth and ZigBee.

During the course of this work, it was used C++ in the Arduino, C in the BDM firmware, and Python in the Text-to-speech sketch. The IAR IDE and Arduino IDE were used for the software development. CadSoft Eagle Cad was used for the PCB design and techniques were applied to the shield layout.

Future work

Some improvements and features can easily be added to the shield, due to the modular programming model used in the BDM firmware and in the Arduino DECT API.

The shield schematic is stable, but the layout can be improved. Some components can be rearranged to make the shield easier to attach to the Arduino. For example, the loudspeaker

connector is on top of the Arduino USB jack, resulting in a slight inclination when the shield is mounted on the Arduino.

Although the BDM supports all DECT functionalities, the current firmware version only contains the most common features, including registration and wireless voice and data transmission. Functionalities such as call handovers, walky-talky mode and dedicated data channels are not yet implemented. These features are application specific and rarely required in small-medium projects. Nonetheless, their implementation and support would be a valuable addition to the shield.

For now, audio reproduction from a file is only supported by a shield working as a fixed part due to limitations in the ENI. A new version of the ENI is required to support portable parts. Streaming and capturing of an audio file is also limited by the ENI and support for more audio codecs can be increased.

Reducing the programming space in the Arduino API is a point to be improved, since it is occupying more than anticipated. Different programming models can be implemented and tested to obtain better results.

Finally, additional tests using more than two devices need to be performed.

REFERENCES

[1] ETS 300-175: Radio Equipment and Systems (RES): Digital Enhanced Cordless Telecommunications (DECT), Parts 1-5, ETSI, 1992.

[2] University of Manchester, "DECT," 2005. [Online]. Available: http://www.cs.manchester.ac.uk/ugt/COMP38512/cs3091_docs/TELE/DECT.html. [Accessed September 2012].

[3] A. Schuler, E. Tews and R.-P. Weinmann, "deDECTed.org," in *25th Chaos Communication Congress (25C3)*, Berlin, Germany, 2008.

[4] Hack n Mod, "Top 40 Arduino Projects of the Web," [Online]. Available: http://hacknmod.com/hack/top-40-arduino-projects-of-the-web/. [Accessed September 2012].

[5] A. Muchaxo, A. Sousa, N. Pereira and H. Sarmento, "Wireless Data Communications Using DECT Air Interface," *ACM SIGCOMM Computer Communication Review,* vol. 29, April 1999.

[6] H. Eriksson, "Performance of dynamic channel allocation in the DECT system," *Vehicular Technology Conference, 1991. Gateway to the Future Technology in Motion., 41st IEEE,* pp. 693 - 698, May 1991.

[7] I. Standard, ISO/IEC 7498-1, Information technology - Open Systems Interconnection - Basic Reference Model: The Basic Model, International Standard, 1994.

[8] A. Glória, "ZBeeMeR," Instituto Superior Técnico, INESC-ID, 2010.

[9] Arduino, "Arduino Official Site," [Online]. Available: http://arduino.cc/. [Accessed May 2012].

[10] Atmel, "megaAVR," [Online]. Available: http://www.atmel.com/products/microcontrollers/avr/megaavr.aspx. [Accessed May 2012].

[11] H. Barragán, "Wiring," [Online]. Available: http://wiring.org.co/. [Accessed May 2012].

[12] J. Oxer, "Arduino Shield List," [Online]. Available: http://shieldlist.org/. [Accessed June 2012].

[13] Arduino, "Arduino Uno," [Online]. Available: http://arduino.cc/en/Main/ArduinoBoardUno. [Accessed May 2012].

[14] ITead Studio, "Stackable Bluetooth Shield," [Online]. Available: http://iteadstudio.com/store/index.php?main_page=product_info&products_id=307. [Accessed June 2012].

[15] Digi, "Digi XBee Wireless RF Modules," [Online]. Available: http://www.digi.com/xbee/. [Accessed June 2012].

[16] Arduino, "Arduino WiFi Shield," [Online]. Available: http://arduino.cc/en/Main/ArduinoWiFiShield. [Accessed August 2012].

[17] B. Landoni, "GSM-GPS shield for Arduino," [Online]. Available: http://www.open-electronics.org/gsm-gps-shield-for-arduino/. [Accessed September 2012].

[18] Micron Electronics, "Sim900 GSM/GPRS+EGSM Module," [Online]. Available:

http://www.simcom.us/product_detail.php?cid=1&pid=37. [Accessed September 2012].

[19] Dialog Semiconductor, "SC14CVMDECT, Cordless Voice Module," 2011.

[20] Dialog Semiconductor, “SC14441, SC14442A, 1.8V Single Chips for DECT with RFPA and QSPI,” 2011.

[21] Microchip, "SPI, Overview and Use of the PICmicro Serial Peripheral Interface," 2002.

[22] Transcend, "microSDHC Card series," 2009.

[23] Freescale, "M68HC11 Reference Manual," 2007.

[24] NXP Semiconductors, "I2C-bus specification and user manual," 2012.

[25] Fairchild Semiconductor, "I2C Translator Pull-up Resisters: Internal vs. External?," 2010.

[26] Texas Instruments, “SN74LVTH125, 3.3-V ABT Quadruple Bus Buffers with 3-State Outputs,” 2003.

[27] Texas Instruments, “SN74AVC4T245, 4-Bit Dual-Supply Bus Transceiver with Configurable Voltage Translation and 3-State Outputs,” 2005.

[28] Philips Semiconductors, “Bi-directional level shifter for I2C-bus and other systems,” 1997.

[29] National Semiconductor, "LM1117/LM1117I, 800mA Low-Dropout Linear Regulator," 2006.

[30] Silicon Labs, "C8051FXXX Printed Circuit Board Design Notes," 2006.

[31] Diodes Incorporated, “74LVC1G125, Single Buffer Gate with 3-State Output,” 2012.

[32] CadSoft, [Online]. Available: http://www.cadsoftusa.com/. [Accessed April 2012].

[33] Eurocircuits, "PCB Proto," [Online]. Available: http://www.eurocircuits.com/images/stories/ec09/ec-pcb-proto-overview-english-1-2010-v3.pdf. [Accessed April 2012].

[34] Cypress, "Top 10 EMC Design Considerations," 2011.

[35] SanDisk, "SanDisk Secure Digital Card," Corporate Headquarters, 140 Caspian Court, Sunnyvale, CA 94089, 2003.

[36] Microsoft, "Microsoft Extensible Firmware Initiative, FAT32 File System Specification," 2000.

[37] J. Capriotti, "FAT32 vs. NTFS," 2000.

[38] Microsoft, "Multimedia Programming Interface and Data Specifications," 1991.

[39] AT&T, “AT&T Natural Voices® Text-to-Speech Demo,” [Online]. Available: http://www2.research.att.com/~ttsweb/tts/demo.php. [Accessed July 2012].

Appendix A SPI TIMING ANALYSIS

In order to obtain high data transfer rates from the SPI bus, it is necessary to determine the maximum SPI clock frequency, f_{clk}, supported by the SPI circuit, that the Arduino, BDM and microSD card can use. The clock frequency depends on:

- t_p^B - Maximum propagation delay of a buffer.
- t_p^T - Maximum propagation delay of a transceiver.
- t_{ddo} - Maximum SPI clock active edge to data-out delay.
- t_{setup} - Minimum SPI data-in setup time.

The minimum period possible of the SPI clock is equivalent to the minimum time the master has to wait for an updated bit on the MISO line after a rising edge on the clock until a falling edge occurs. The calculation of this value starts by taking into consideration the time required for the clock signal to go from the master to the slave, t_{pclk}. In this project, this value is given by (1) and consists of the propagation delay of each buffer and transceiver the signal needs to go through.

$$t_{pclk} = N_{Buffers} \times t_p^B + N_{Transceivers} \times t_p^T \quad (1)$$

After the slave detects the clock signal, the time it takes for the slave to update the data-out bit needs to be taken into consideration, t_{ddo} . This new bit needs to reach the master and also takes additional time, t_{pmiso}. The bit path might be formed by some buffers and possibly a transceiver, so the formula for t_{pclk} is the same for t_{pmiso}. There is also a setup time associated to the data-in line, the new bit needs to be detected by the master at least t_{setup} before a falling edge.

The final formula to calculate the maximum SPI clock is given by (2).

$$f_{clk} = \frac{1}{t_{clk}} = \frac{1}{2 \times t_{min}} = \frac{1}{2 \times \left(t_{pclk} + t_{ddo} + t_{pmiso} + t_{setup}\right)} \quad (2)$$

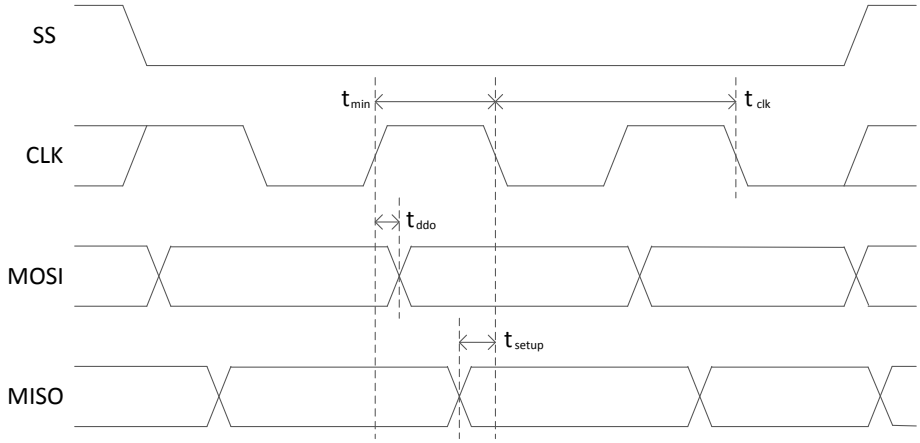
The time necessary for the MOSI signal to go from the master to the slave is not taken into account, because it normally takes more time than t_{pclk}, but less than $t_{min} - t_{setup}$. In all three possible SPI communication channels, this condition is true and it does not affect the clock speed.

Table A.1 shows the timings, for each IC present in the shield, and Figure A.1 illustrates the SPI protocol in Mode 2. In this mode, the input bits are sampled on the clock falling edges and are updated on the rising edges. The Arduino, BDM and microSD use Mode 2 for communication.

Table A.1 - Time constraints for each device in nanoseconds.

Time	Arduino	Bithium DECT Module		microSD [22]	SN74LVTH125 [26]	SN74AVC4T245 [27]
		Master	Slave			
t_{ddo}	10ns	5ns	12.5ns	14ns	-	-
t_{setup}	10ns	12.4ns	5ns	6ns	-	-
t_p	-	-	-	-	3.9ns	4.5ns

Figure A.1 - SPI signals in Mode 2.



The first maximum SPI clock calculated is for when the Arduino is communicating with the BDM. In this case the Arduino is in master operation and the module is in slave. The clock signal needs to go through one buffer and one transceiver and the data-out line (DO)from the BDM goes through two buffers and one transceiver. Using the timings present in Table A.1 and the three equation system formed by (3), (4) and (5), the maximum clock speed is calculated.

$$f_{clk}^{A,M} = \frac{1}{2 \times (8.4ns + 12.5ns + 12.3ns + 10ns)} \approx 11.6MHz \quad (3)$$

$$t_{pclk}^{A,M} = t_p^B + t_p^T = 3.9ns + 4.5ns = 8.4ns \quad (4)$$

$$t_{pmiso}^{M,A} = 2 \times t_p^B + t_p^T = 2 \times 3.9ns + 4.5ns = 12.3ns \quad (5)$$

Since the SPI communication channel between the Arduino and the module allows speeds up to 11.6 MHz and the Arduino SPI interface can work at a maximum frequency of 8 MHz, it means that there should be no issues if the maximum frequency is used.

The second clock calculated is for when the Arduino is accessing the memory card. Both the clock signal and the microSD DO only need to go through one buffer. No transceivers are used in this communication channel. Equations (6), (7) and (8) form the system to calculate $f_{clk}^{A,SD}$.

$$f_{clk}^{A,SD} = \frac{1}{2 \times (3.9ns + 14ns + 3.9ns + 10ns)} \approx 15.7MHz \quad (6)$$

$$t_{pclk}^{A,SD} = t_p^B = 3.9ns \quad (7)$$

$$t_{pmiso}^{SD,A} = t_p^B = 3.9ns \quad (8)$$

The full bandwidth of the Arduino SPI interface can be used because the channel supports frequencies up to 15.7 MHz.

Finally, the last maximum SPI clock frequency is calculated using (9), (10) and (11) and is for when the DECT module is communicating with the microSD card. In this case, the clock signal coming from the BDM goes through one transceiver and the DO pin from the microSD through a single buffer.

$$f_{clk}^{M,SD} = \frac{1}{2 \times (4.5ns + 14ns + 3.9ns + 12.4ns)} \approx 14.7MHz \quad (9)$$

$$t_{pclk}^{M,SD} = t_p^T = 4.5ns \quad (10)$$

$$t_{pmiso}^{SD,M} = t_p^B = 3.9ns \quad (11)$$

Although the module can output a clock frequency up to 20 MHz, the channel only support a maximum of 14.7MHz, which is better than using SPI emulation. This clock frequency proved to be enough to read and write an audio file quickly with average bitrates.

Appendix B PRODUCTION DOSSIER

This appendix contains the Production Dossier which includes the information to produce and assemble the DECT Shield (Rev. B).

Firstly, the circuit schematic is presented in section B.1. The drawings were made using the Light version of Eagle CAD 6.1.0 software from CadSoft [32]. Secondly, the Bill of Materials (BOM) is shown on Table B.1 of section B.2. Prices for the necessary components were gathered from Farnell, Digikey and Mouser. Finally, on section B.3, the PCB layout artwork extracted from the Gerber files (RS274X) is presented.

B.1 SCHEMATIC

Figure B.1 - DECT Shield schematic (1 of 2).

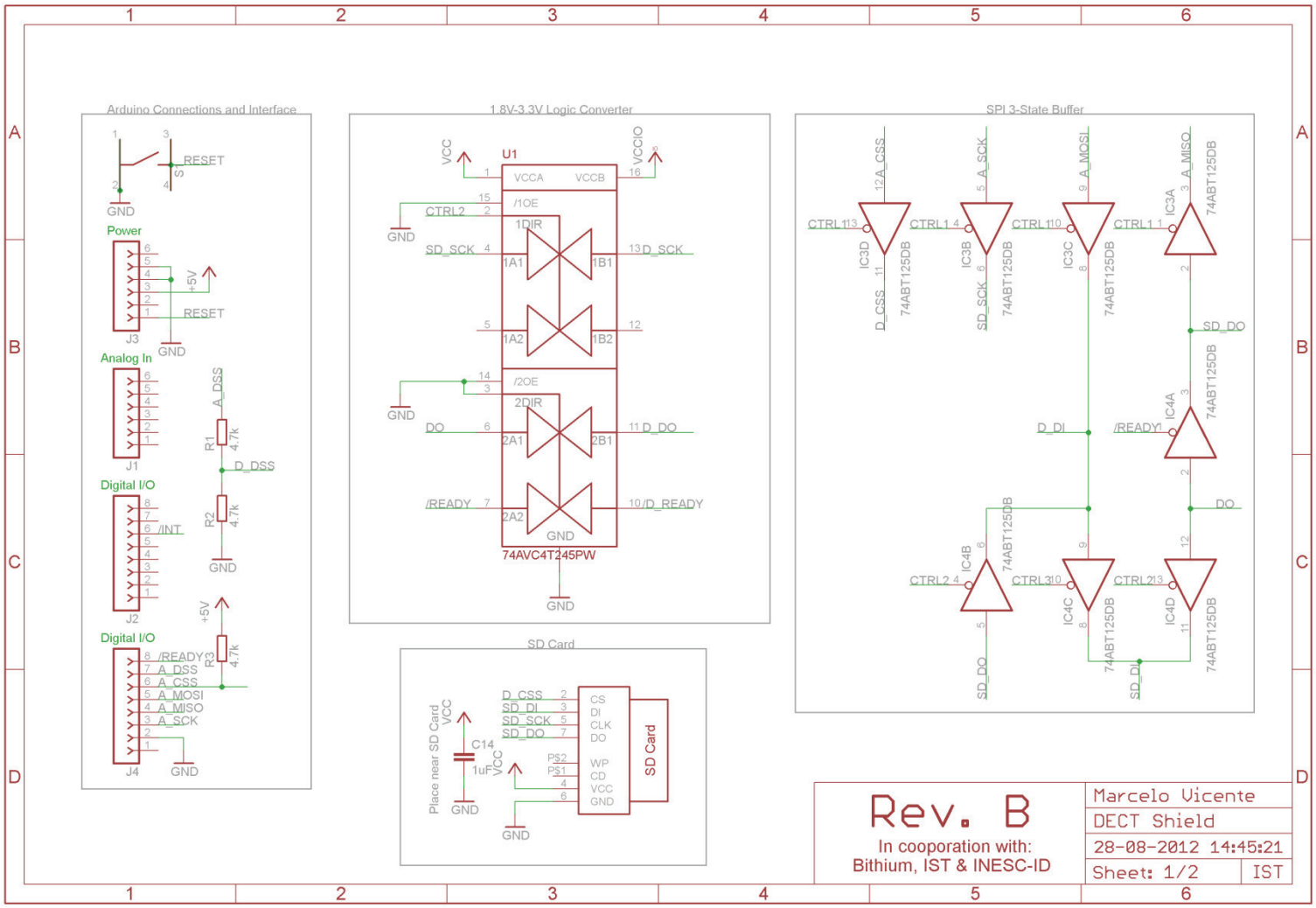
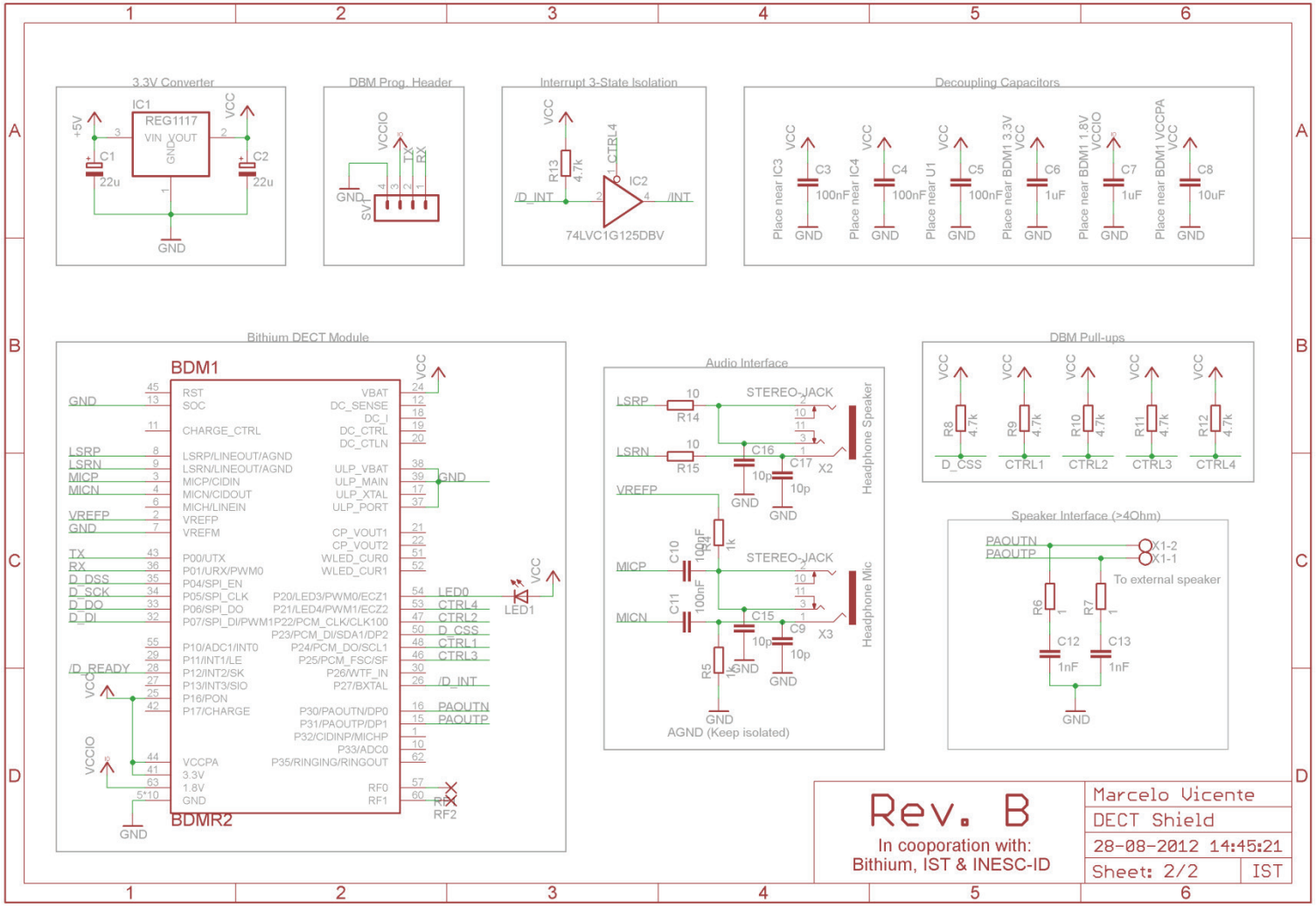


Figure B.2 - DECT Shield schematic (2 of 2).



B.2 BILL OF MATERIALS

Table B.1 - Bill of Materials.

Item	Qty	Value	Device	Parts	Manufacturer	Manufacturer Reference
1	2	1	R-EU_R0603	R6, R7	PANASONIC	ERJ3GEYJ1R0V
2	2	10	R-EU_R0603	R14, R15	PANASONIC	ERJ3GEYJ100V
3	2	1k	R-EU_R0603	R4, R5	PANASONIC	ERJ3GEYJ102V
4	9	4.7k	R-EU_R0603	R1, R2, R3, R8, R9, R10, R11, R12, R13	PANASONIC	ERJ3GEYJ472V
5	2	22u	CPOL-EUUD-6,3X8,8	C1, C2	MULTICOMP	MCESL35V226M6.3X5.2
6	1	10uF	C-EUC0805	C8	MULTICOMP	MCCA000268
7	3	1uF	C-EUC0603	C6, C7, C14	MULTICOMP	MCCA000530
8	5	100nF	C-EUC0603	C3, C4, C5, C10, C11	MULTICOMP	MCCA000256
9	2	1nF	C-EUC0603	C12, C13	MULTICOMP	MCCA000224
10	4	10p	C-EUC0603	C9, C15, C16, C17	MULTICOMP	MCCA000192
11	1	RED	LEDSML0805	LED1	KINGBRIGHT	KPT-2012SURCK
12	2	74ABT125DB	74ABT125DB	IC3, IC4	TEXAS INSTRUMENTS	SN74LVTH125DBR
13	1	74AVC4T245PW	74AVC4T245PW	U1	TEXAS INSTRUMENTS	SN74AVC4T245PW
14	1	74LVC1G125DBV	74LVC1G125DBV	IC2	DIODES INC.	74LVC1G125W5-7
15	1	REG1117	REG1117	IC1	NATIONAL SEMICONDUCTOR	LM1117IMP-3.3/NOPB
16	1	BDMR2	BDMR2	BDM1	BITHIUM	BITHIUM DECT MODULE
17	2	STEREO-JACK	STEREO-JACK	X2, X3	SWITCHCRAFT	35RAPC4BHN2
18	1	SDCARDM	SDCARDM	U2	HRS (HIROSE)	DM3D-SF
19	1	RESET	SWITCH	S1	TE CONNECTIVITY	FSM2JH
20	1	TERMBLOCK	W237-102	X1	CAMDEN	CTB1202/2
21	2	-	F-1X06-SIP-100-40	J1, J3	SAMTEC	ESQ-106-14-T-S
22	2	-	F-1X08-SIP-100-40	J2, J4	SAMTEC	ESQ-108-14-T-S
23	1	-	MA04-1	SV1	MULTICOMP	MC34749

Table B.2 - Component prices from Farnell, Digikey and Mouser[5].

Item	Farnell Reference	Farnell Price	Farnell Min. Quantity	Digikey Reference	Digikey Price	Digikey Min. Quantity	Mouser Reference	Mouser Price	Mouser Min. Quantity
1	2059528	0.01€	50	P1.0GCT-ND	0.09€	1	667-ERJ-3GEYJ1R0V	0.08€	1
2	2059548	0.01€	50	P10GCT-ND	0.09€	1	667-ERJ-3GEYJ100V	0.08€	1
3	2059601	0.01€	50	P1.0KGCT-ND	0.09€	1	667-ERJ-3GEYJ102V	0.08€	1
4	2059619	0.01€	50	P4.7KGCT-ND	0.09€	1	667-ERJ-3GEYJ472V	0.08€	1
5	9265724	0.10€	10	N/A	N/A	N/A	N/A	N/A	N/A
6	1759136	0.04€	100	N/A	N/A	N/A	N/A	N/A	N/A
7	1759408	0.01€	100	N/A	N/A	N/A	N/A	N/A	N/A
8	1759123	0.01€	100	N/A	N/A	N/A	N/A	N/A	N/A
9	1759088	0.01€	100	N/A	N/A	N/A	N/A	N/A	N/A
10	1759053	0.01€	100	N/A	N/A	N/A	N/A	N/A	N/A
11	2099241	0.09€	1	N/A	N/A	N/A	604-APHCM2012SURCK	0.18€	1
12	1236439	0.23€	1	296-8658-1-ND	0.57€	1	595-SN74LVTH125DBR	0.42€	1
13	1053499	0.80€	1	296-18056-1-ND	0.98€	1	595-SN74AVC4T245PW	0.83€	1
14	1893827	0.10€	1	74LVC1G125W5-7DICT-ND	0.35€	1	621-74LVC1G125W5-7	0.34€	1
15	1469051	0.76€	1	LM1117IMP-3.3CT-ND	1.00€	1	926-LM1117IMP3.3NOPB	1.01€	1
16[6]	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A
17	1608405	0.76€	1	SC1459-ND	1.23€	1	502-35RAPC4BHN2	0.89€	1
18	1764377	1.59€	1	HR1941CT-ND	1.83€	1	798-DM3D-SF	1.67€	1
19	1555981	0.12€	1	450-1649-ND	0.18€	1	506-FSM2JH	0.18€	1
20	1716993	0.27€	1	N/A	N/A	N/A	N/A	N/A	N/A
21	1927510	1.43€	10	N/A	N/A	N/A	N/A	N/A	N/A
22	1928229	1.90€	10	N/A	N/A	N/A	N/A	N/A	N/A
23	1593428	0.08€	10	N/A	N/A	N/A	N/A	N/A	N/A

[5] Farnell, Digikey and Mouser information regarding references, prices and minimum quantity were obtained in July of 2012.
[6] Can be bought directly from Bithium and cost approximately 15€.

B.3 LAYOUT

Figure B.3 - Bottom copper layer

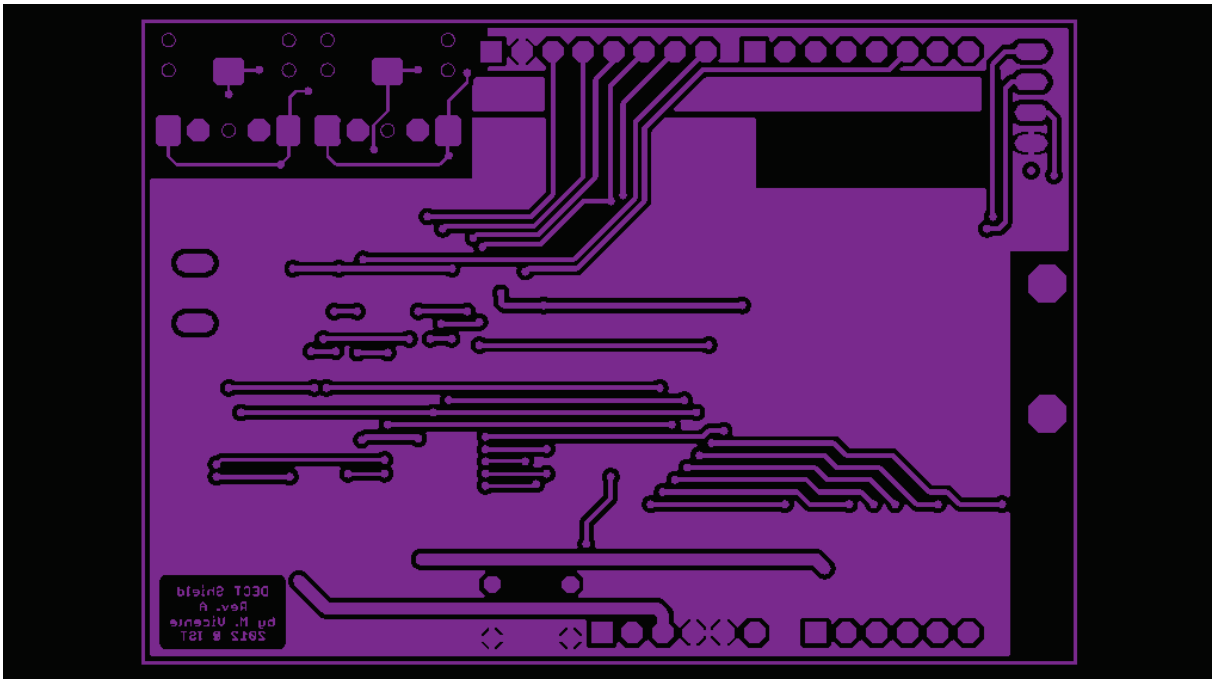


Figure B.4 - Bottom solder mask

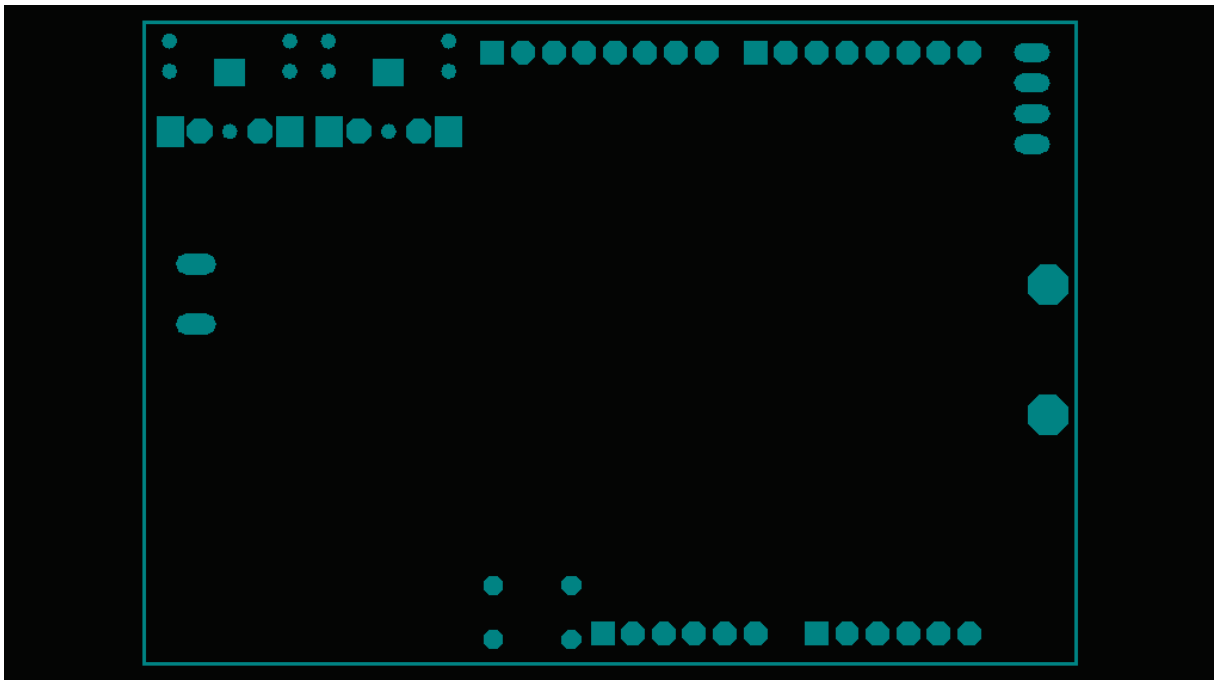


Figure B.5 - Top copper layer

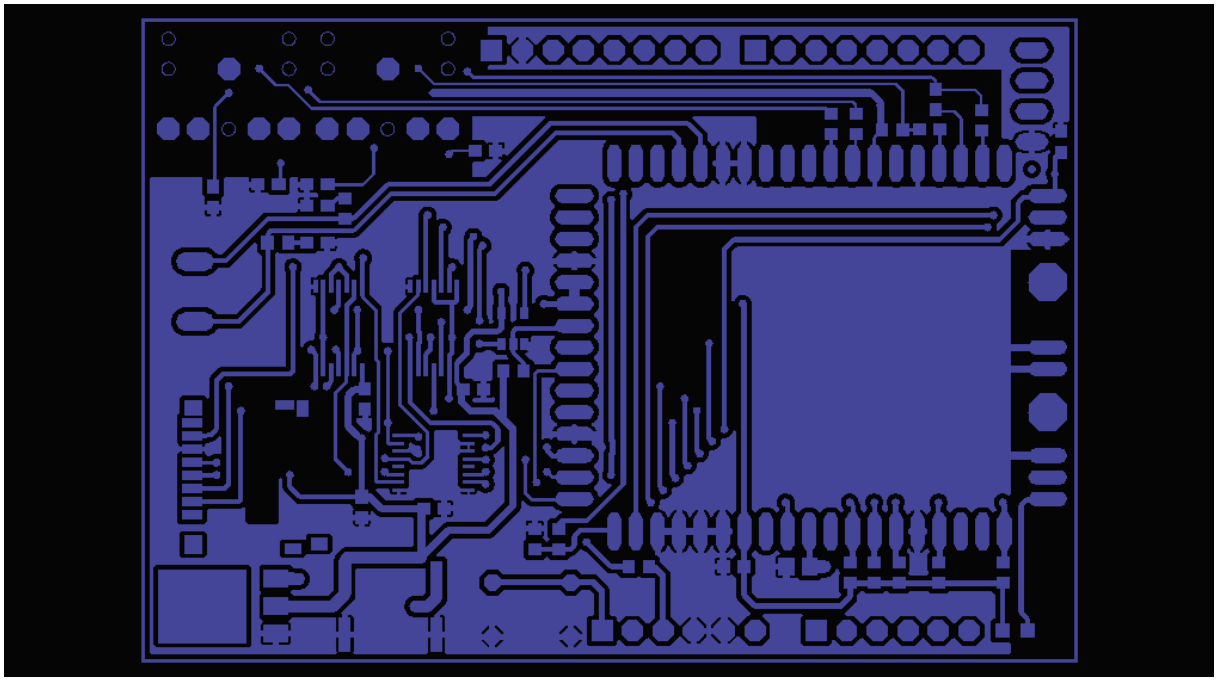


Figure B.6 - Top solder mask

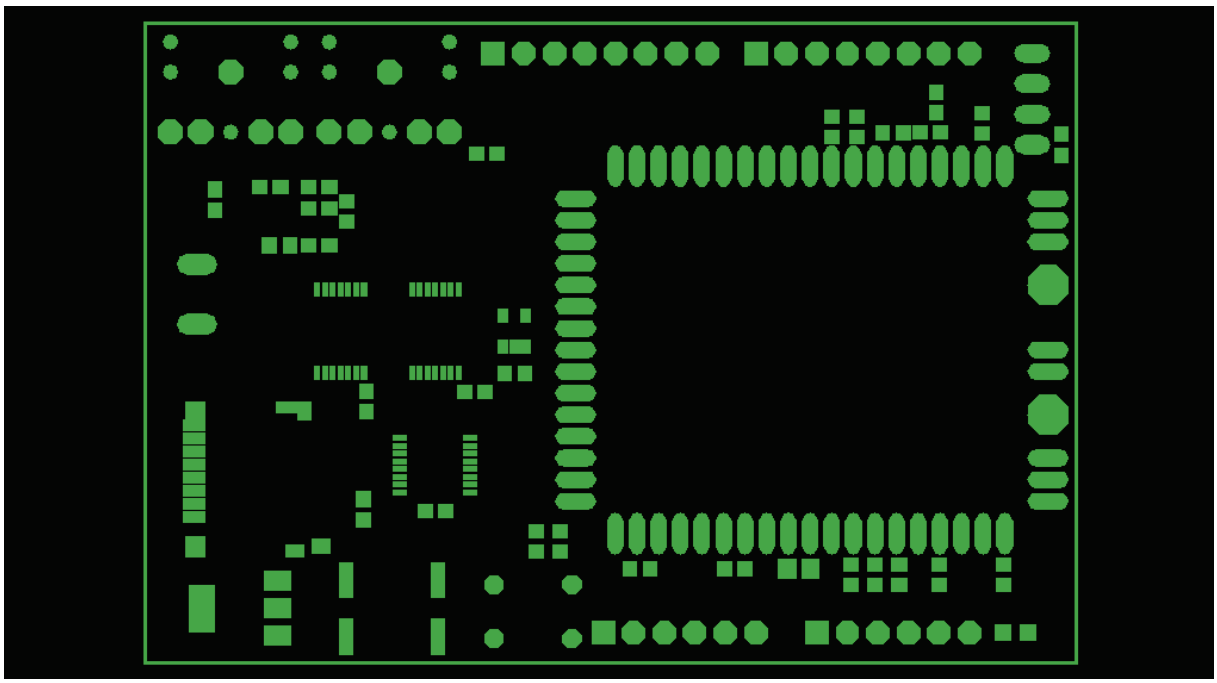


Figure B.7 - Top silkscreen layer

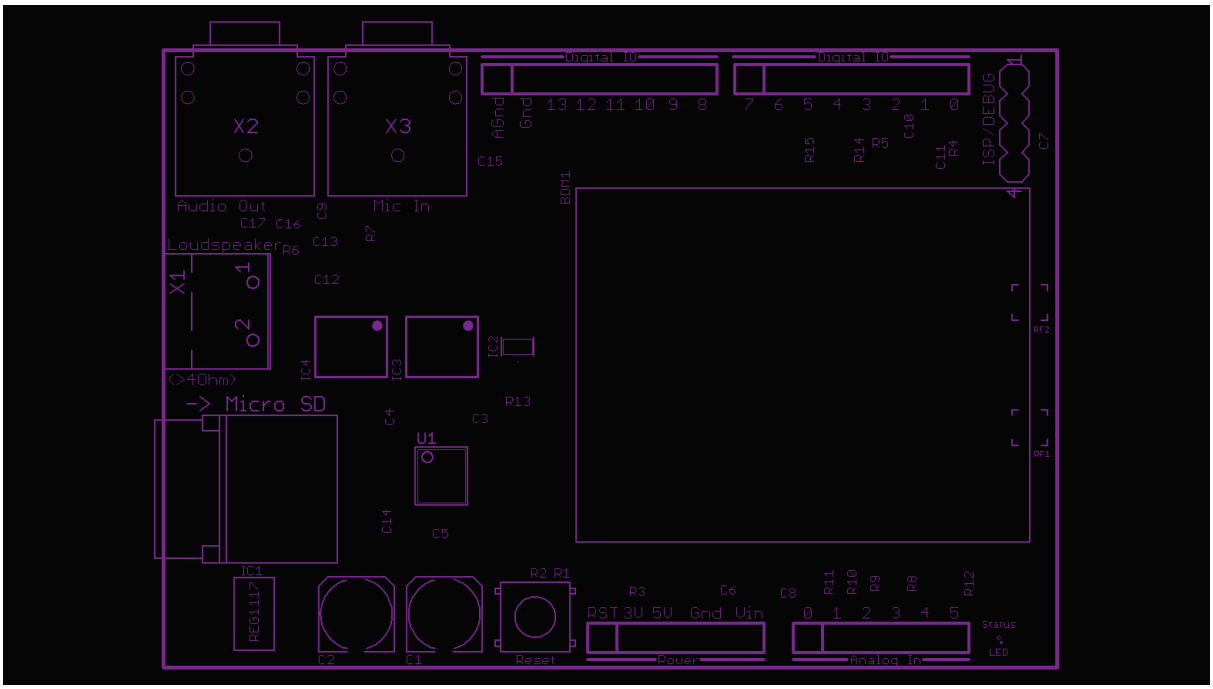
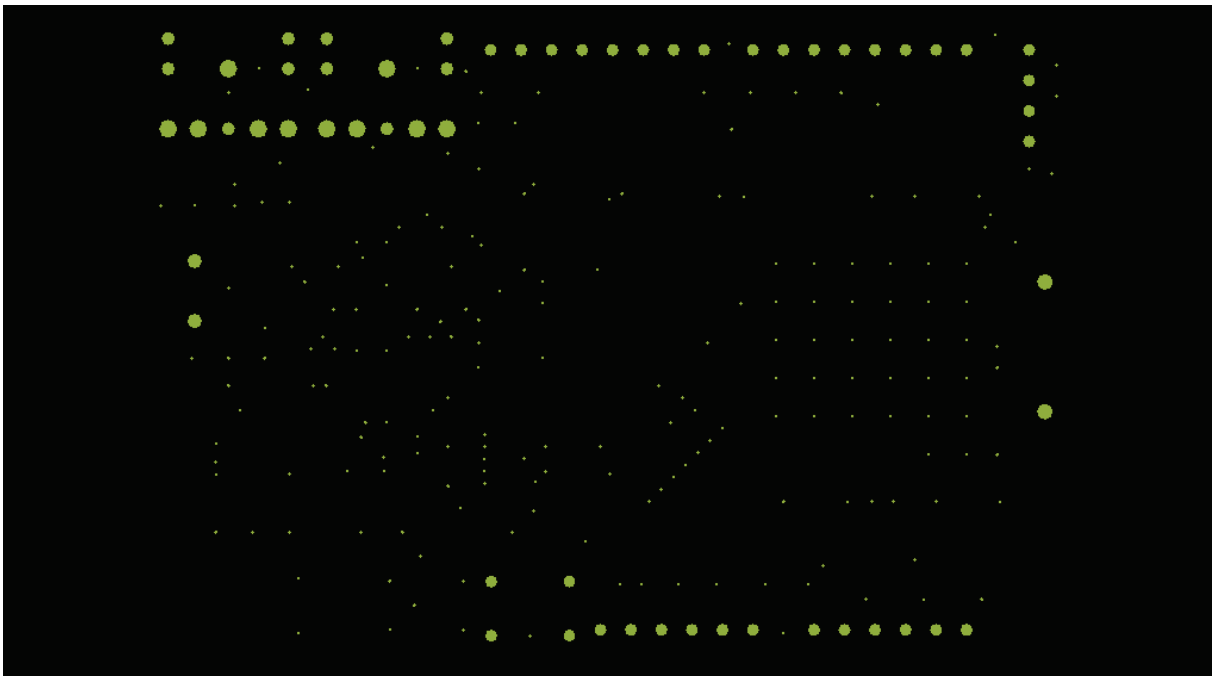


Figure B.8 - Drill plan



Appendix C EXTERNAL INTERFACE LAYER

This appendix contains the description of the commands supported by the External Interface Layer (EIL). The commands are sent by the Arduino, using messages from the Protocol layer, to interact with the DECT Shield. The EIL processes and executes the commands in the Bithium DECT Module. Usage of the EIL is demonstrated in the source code of the DECT API for Arduino.

Table C.1 lists all commands supported by the EIL. The current software version defines 34 different commands and can be extended up to 255. The commands are grouped in families and each family is targeted to a specific set of functionalities supported by the shield.

Table C.1 - List of commands grouped in families.

Command Families				
Module	**DECT**	**Playback**	**Audio**	**microSD**
moduleVersion moduleFirmware moduleReset moduleDefaults moduleStatus moduleConfig moduleEvent	dectInfo dectPincode dectList dectSend dectRecv dectCall dectCalllist dectRegister dectUnregister dectScan dectScanlist dectStatus	playbackPlay playbackStop playbackVolume playbackStatus playbackOpen playbackSize playbackQueue	audioSpeaker audioLoudspeaker audioVolume audioMic audioGain audioConfig	sdcardAccess sdcardRelease

Table C.2 presents the description of commands in the Module family. These commands are used to control or get information about the BDM, such as the firmware version, and the configurations.

Table C.2 - Commands of the Module family.

ID	Command	Description
0	moduleVersion	Returns the current firmware running in the module as well its current version in the following format: (FP/FP, Major Revision, Minor Revision).
1	moduleFirmware	Changes the module firmware to Fixed Part or Portable Part.
2	moduleReset	Performs a software reset of the module.
3	moduleDefaults	Same as moduleReset, but default settings are also applied during reset.
4	moduleStatus	Returns the module current status information, this also includes status for the DECT, Playback and Audio.
5	moduleConfig	Returns or sets the module configurations based on the arguments. These also include the DECT and Audio configurations.
6	moduleEvent	Returns the oldest pending event (if available) from the module.

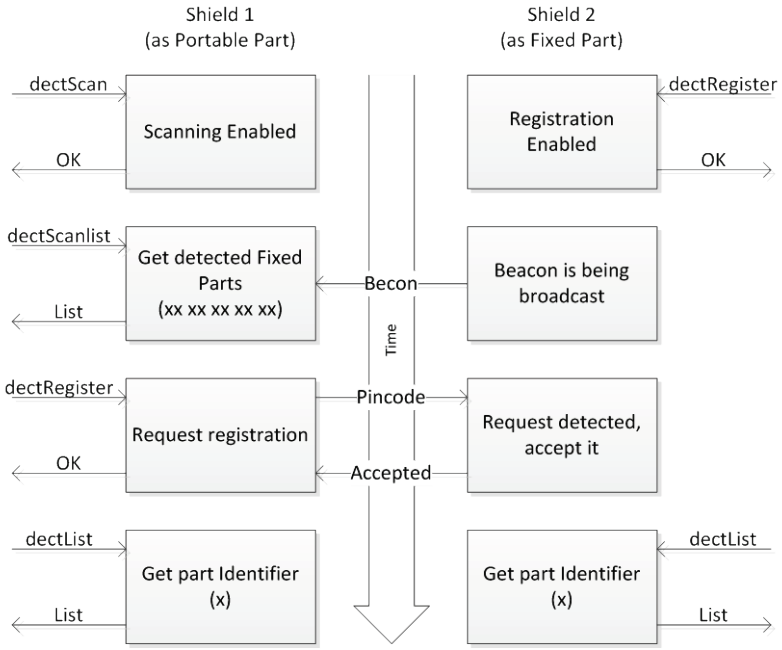
DECT commands are presented in Table C.3. Some commands have dual functionality, depending on the BDM firmware. The dectScan and dectScanlist commands, which are used to scan for Fixed Parts, are only accepted by Portable Parts.

Table C.3 - Commands of the DECT family.

ID	Command	Description
16	dectInfo	Fixed Part: Returns the module's DECT RFPI. Portable Part: Returns the module's DECT IPUI.
17	dectPincode	Change or returns the module's DECT current pin-code.
18	dectList	Returns the list of known/registered fixed/portable parts.
19	dectSend	Send a chunk of data (up to 54 bytes) to a target shield.
20	dectRecv	Read one data chunk from the receiver FIFO structure.
21	dectCall	Make, accept, refuse or terminate a call to/from another shield.
22	dectCalllist	Returns the list of all call channels indicating the state of each call.
23	dectRegister	Fixed Part: Enable or disable shield registration mode in the shield. Portable Part: Register module to a target Fixed Part.
24	dectUnregister	Fixed Part: Unregister a Portable Part from the module. Portable Part: Unregister the module from the current Fixed Part.
25	dectScan	Portable Part: Enable or disable scanning mode for Fixed Parts.
26	dectScanlist	Portable Part: Returns the list of Fixed Parts that were detected in range.
27	dectStatus	Returns the current DECT status information from the module.

Registration between shields requires at least one working as fixed part and another as portable part. The fixed part enables registration using the dectRegister command and the portable part can scan for base stations using the dectScan and dectScanlist commands. The portable part registers to the fixed part using the dectRegister command. Both shields need to have the same pincode, which can be set using the dectPincode command. Figure C.1 illustrates the registration procedure.

Figure C.1 - Shield registration procedure.



After registration, there is an identifier associated to the target shield, which can be obtained using the dectList command. dectRecv, dectSend and dectCall can use the identifiers to reference a shield. Identifier 0 represents all registered parts and can be used to send a data packet to every registered module, for example.

The commands to control audio file reproduction in the shield are listed in Table C.4. Audio playback is restricted to shields working as fixed parts and playback commands sent to portable parts are rejected.

Table C.4 - Commands of the Playback family.

ID	Command	Description
48	playbackPlay	Plays a previously opened WAV file or a sequence of files.
49	playbackStop	Stops playing the current file (if any).
50	playbackVolume	Changes the playback volume directly in the DSP from 0 to 15.
51	playbackStatus	Returns the current playback status information from the module.
52	playbackOpen	Open a file from the memory card (if compatible).
53	playbackSize	Changes the playback queue size.
54	playbackQueue	Queue a file from the memory card to the playlist (if compatible).

Audio files can be opened using the playbackOpen or queued using the playbackQueue command. The playbackPlay command begins reproducing the previously opened file or assembled queue. The queue size needs to be set using playbackSize before issuing playbackQueue commands.

Table C.5 presents the commands to control audio components, such as speaker volume and microphone gain.

Table C.5 - Commands of the Audio family.

ID	Command	Description
64	audioSpeaker	Enable or disable the shield's speaker connection.
65	audioLoudspeaker	Enable or disable the shield's loudspeaker connection.
66	audioVolume	Change the speaker/loudspeaker volume from 2dB (0) to -12dB (7) in steps of -2dB.
67	audioMic	Enable or disable the shield's microphone connection.
68	audioGain	Change the microphone gain from 0dB (0) to 30dB (15) in steps of 2dB.
69	audioConfig	Returns the current audio configurations from the module.

Finally, the last two commands present in Table C.6, permit the Arduino to control the microSD card. The shield is not able to reproduce audio files when the Arduino has access to the card, therefore a sdcardRelease command must be issued before executing any playback operation. The shield is booted with the card released.

Table C.6 - Commands of the microSD family.

ID	Command	Description
80	sdcardAccess	Allows the Arduino to access the memory card.
81	sdcardRelease	Releases the memory card from the Arduino.

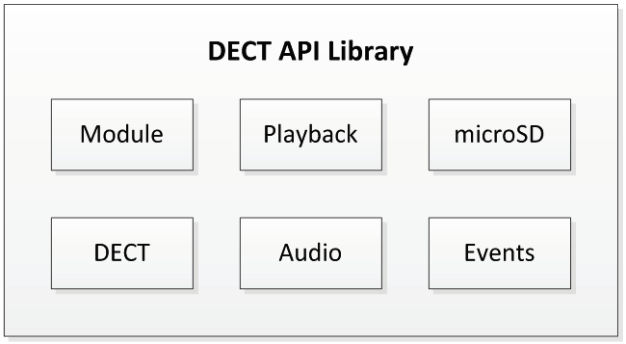
Appendix D DECT API

This appendix contains the documentation for the DECT API library (Rev. A), used to interface the Arduino with the DECT Shield. Firstly, it presents an overview to explain the basic concepts and operation. Afterwards, the full function set is listed and a detailed description of each function is provided.

D.1 LIBRARY OVERVIEW

The DECT API is a library developed in C++ targeted for the Arduino. This library implements the interface between the Arduino and the DECT Shield. The library functions are separated in six groups as illustrated in Figure D.1.

Figure D.1 - DECT API library groups.



Each group deals with different functionalities supported by the shield:

- Module - Configuration and status.
- DECT - Control of calls, data transmission and registration.
- Playback - Reproduction of files stored in the microSD card.
- Audio - Configuration of the speaker, loudspeaker and microphone.
- microSD - Control of the memory card access.
- Events - Configuration, handling and triggering of events.

D.2 LIBRARY USAGE

Figure D.2 presents the main points to use the library in an Arduino sketch. Firstly, the library is initialized by creating a global instance of the DECT class, where the communication ports used by the

shield are initialized. Secondly, the shield is configured in the setup function (e.g. speaker volume, DECT pincode). Finally, control code (e.g. make call, analyze data payload) can be executed in the sketch loop function to manage the shield components.

Figure D.2 - DECT API library sketch flow.

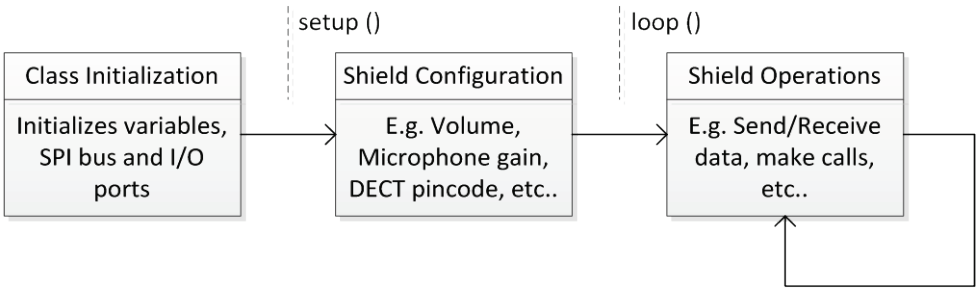


Figure D.3 shows an empty sketch with the necessary code to use the DECT Shield. The first instruction creates an instance of the DECT class. The argument with the “false” value indicates that events are not used in the sketch.

To configure the shield, it is necessary to call the `GetConfigs` function in order to retrieve the configuration structure from the shield. The structure contains the current configurations and can be changed to obtain the intended shield behavior. New configurations are applied after sending the structure back to the shield using the `SetConfigs` function.

Figure D.3 - DECT Shield empty sketch for Arduino.

```
// DECT Shield Empty Sketch
#include <DECT.h>

// DECT class WITHOUT events, true for events
DECT dect(false);

void setup() {
  stConfigs sConfigs;
  dect.GetConfigs(&sConfigs);
  // TODO: Change shield configurations
  dect.SetConfigs(&sConfigs);

  // TODO: Apply other configurations
}

void loop() {
  // TODO: Add control code here
}
```

For a simple DECT link between two shields, one needs to be configured as a fixed part and the other as a portable part. Firmware configurations are persistent and can be applied using the `SetFirmware` function.

Before any audio or data transference, the shield working as portable part needs be registered to a fixed part. The fixed part can enable registration using the `Register` function and the portable part can use the `Scan` function to start scanning for fixed parts. The `GetScanList` function is used afterwards to retrieve the list of fixed parts which were found. Registration is established when the portable part executes the `Register` function with the target fixed part identifier, retrieved from `GetScanList`. Both devices need to have the same pin code which can be set using the `SetPincode` function. At any time, the fixed and portable parts can use the `GetList` function to retrieve the identifiers of all registered parts. These identifiers are used with the DECT functions to reference other DECT devices. Identifier 0 represents all registered parts.

The library provides functions to send and receive data packets with up 54 bytes. The `Send` function is used to transmit a data packet to a target DECT device. To read an incoming data packet, it is necessary to use the `Recv` function. The total number of incoming packages present in the shield can be retrieved using the `GetStatus` or `GetDECTStatus` functions. The library can be configured to generated and event when a new packet arrives, automatically transferring the packet to the Arduino. Figure D.4 contains a code snippet exemplifying data transference.

Figure D.4 - Arduino exemplification code for data transference.

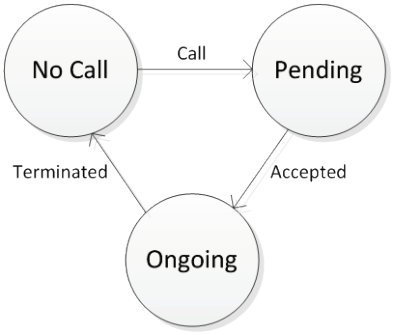
```
void Loop() {
  // ...
  // Send a string to all registered DECT devices
  char pString[54] = "Hello!";
  dect.Send(0, pString, strlen(szString)+1);
  // ...
  // Read an incoming packet
  uint8_t nIdentifier; // Sender identifier
  uint8_t nPacketSize = dect.Recv(&nIdentifier, pString, sizeof(pString));
  // ...
```

Calls are handled using the `Call` and `GetCallList` functions. `GetCallList` returns a list of the voice communication status (or calls) of each registered part using the identifiers obtained from `GetList`. The calls can have three different states:

- No Call, no call established. When a call is refused or terminated it goes to this state;
- Pending, waiting for an answer. If the call is outgoing, then it is waiting for the destination to answer, otherwise it is waiting for the user answer. A call in this state can be refused or accepted;
- Ongoing, communication established. The audio channel between the two devices is formed, voice communication is possible.

Figure D.5 illustrates the normal state sequence of a call.

Figure D.5 - DECT call states sequence.



The No Call state can have multiple variants which transmit how the call was terminated:

- Refused, call was refused.
- Terminated by User, call was terminated by the used.
- Terminated by Destination, call was terminated by destination.
- Disconnected, call was lost (e.g. out of range).
- Invalid. Unhandled error, should never happen.

The `Call` function is used to make, accept and refuse calls which in turn change the call states. The function is capable of performing one of four actions:

- Make call, attempt to make a call with the target registered part;
- Accept call, accept an incoming call;
- Refuse call, refuse an incoming call;
- Terminate call, terminate an ongoing call.

A call to all registered parts can be executed using the identifier 0. The first part to answer the call becomes the endpoint.

Figure D.6 contains a code snippet exemplifying on how to make a call and wait for the answer of the target device.

Figure D.6 - Arduino exemplification code for making calls.

```
void Loop() {
  // ...
  // Call a registered part and wait for an answer
  stCallElem sListElem;
  dect.Call(1, MakeCall);
  do {
    // Only retrieve call slot 0
    dect.GetCallList(&sListElem, 1);
  } while (sListElem.eState == Pending);
  if (sListElem.eState == Ongoing)
    Serial.println("Call was accepted!");
  else
    Serial.println("Call was refused!");
  // ...
}
```

D.3 MODULE GROUP FUNCTIONS

This group includes the following functions:

- `enum DECTError LastError(void);`
- `stVersion Version(void);`
- `void Reset(void);`
- `void Defaults(void);`
- `bool GetConfigs(stConfigs *pConfigs);`
- `bool SetConfigs(stConfigs *pConfigs);`
- `stStatus GetStatus(void);`

The description of each structure and function present in this group follows.

Structure:	`enum DECTError`
Description:	Represent an error with the shield communication.

```
enum DECTError {
  NoError = 0,         // No error occurred
  InvalidCommand,      // Invalid command ID
  InvalidChecksum,     // Checksum mismatch
  InvalidTermination, // Message termination problem
  Timeout,             // Timeout
  BufferTooSmall,      // Given buffer doesn't have enough space
  ReadyNotDeasserted, // SPI CS not deasserted
  ReadyNotAsserted     // SPI CS not asserted
};
```

Structure:	`struct stVersion`
Description:	Shield version structure.

```
typedef struct {
       uint8_t nMinor    :5; // Minor Version
       uint8_t nMajor    :2; // Major Version
       uint8_t nFirmType :1; // Firmware Type
} stVersion;
```

Remarks:	`nFirmType` filed is 0 for Fixed Part firmware or 1 for Portable Part.

Structure:	`struct stConfigs`
Description:	Configuration structure for every component in the shield.

```
typedef struct {
  stEventConfigs sEvents;      // Event configurations
  stDECTConfigs sDECT;         // DECT configurations
  stPlaybackConfigs sPlayback; // Playback configurations
  stAudioConfigs sAudio;       // Audio configurations
} stConfigs;
```

Remarks:	The structures forming `stConfigs` are described below.

Structure:	struct stDECTConfigs
Description:	DECT configuration structure.

```
typedef struct {
  uint8_t bAutoAcceptCalls :1; // Auto accept incoming calls
  uint8_t _res             :7; // Reserved
} stDECTConfigs;
```

Remarks:	Present in stConfigs.

Structure:	struct stEventConfigs
Description:	Event configuration structure.

```
typedef struct {
  uint8_t bInterrupts :1;         // Enable interrupt pin
  uint8_t _res :7;                // Reserved
  union {
    uint8_t nEventMask;           // Event mask
    struct {
      uint8_t bCallEvents     :1; // Call events Enabled
      uint8_t bDataEvents     :1; // Enable Data Events
      uint8_t bPlaybackEvents :1; // Enable Playback Events
      uint8_t _res2           :6; // Reserved
    };
  };
} stEventConfigs;
```

Remarks:	Used in stConfigs.

Structure:	struct stPlaybackConfigs
Description:	Playback configuration structure.

```
typedef struct {
  uint8_t nPrioMode :2;        // Priority Access
  uint8_t _res :6;             // Reserved
} stPlaybackConfigs;
```

Remarks:	Present in stConfigs.

Structure:	struct stAudioConfigs
Description:	Audio configuration structure.

```
typedef struct {
  stAudioSpeaker sSpeaker; // (Loud)Speaker configurations
  stAudioMic sMic;         // Microphone configurations
} stAudioConfigs;
```

Remarks:	The structures forming stAudioConfigs are described below.

Structure:	`struct stAudioSpeaker`
Description:	Speaker and loudspeaker configuration structure.

```
typedef struct {
  uint8_t nVolume             :3; // (Loud)Speaker volume
  uint8_t _res                :3; // Reserved
  uint8_t bLoudspeakerEnabled :1; // Loudspeaker Enabled
  uint8_t bEnabled            :1; // Speaker Enabled
} stAudioSpeaker;
```

Remarks:	Present in `stAudioConfigs`; Speaker volume goes from 0 (-2dB) to 7 (12dB) in steps of 2dB.

Structure:	`struct stAudioMic`
Description:	Microphone configuration structure.

```
typedef struct {
  uint8_t nGain    :4; // Microphone Gain
  uint8_t _res     :3; // Reserved
  uint8_t bEnabled :1; // Microphone Enabled
} stAudioMic;
```

Remarks:	Present in `stAudioConfigs`; Microphone gain goes from 0 (0dB) to 15 (30dB) in steps of 2dB.

Structure:	`struct stStatus`
Description:	Shield status structure.

```
typedef struct {
  stEventStatus sEvents;      // Event Status
  stCallStatus  sCalls;       // Call Status
  stDataStatus  sData;        // Data Status
  stPlaybackStatus sPlayback; // Playback Status
} stStatus;
```

Remarks:	The structures forming `stStatus` are described below.

Structure:	`struct stEventStatus`
Description:	Event status structure.

```
typedef struct {
  uint8_t nCounter; // Number of pending events
} stEventStatus;
```

Remarks:	Present in `stStatus`.

Structure:	struct stDECTStatus
Description:	DECT status structure.

```
typedef struct {
  stDataStatus sData;  // DECT Data Status
  stCallStatus sCalls; // DECT Call Status
} stDECTStatus;
```

Remarks:	Present in stStatus. The structures forming stStatus are described below.

Structure:	struct stDataStatus
Description:	DECT data status structure.

```
typedef struct {
  uint8_t nDataPackets     :3; // Number of available data packets
  uint8_t _res             :5; // Reserved
} stDataStatus;
```

Remarks:	Present in stDECTStatus.

Structure:	struct stCallStatus
Description:	DECT call status structure.

```
typedef struct {
  uint8_t nOngoingCalls    :3; // Number of ongoing calls
  uint8_t nPendingCalls    :3; // Number of pending calls
  uint8_t _res             :1; // Reserved
  uint8_t bRegistrationMode:1; // Registration mode is Enabled
} stCallStatus;
```

Remarks:	Present in stDECTStatus.

Structure:	struct stPlaybackStatus
Description:	Playback status structure.

```
typedef struct {
  uint8_t bPlaying :1; // File is being played
  uint8_t _res     :5; // Reserved
  uint8_t bUpdated :1; // Data block updated
  uint8_t bValid   :1; // Opened file is valid
} stPlaybackStatus;
```

Remarks:	Present in stStatus.

Function:	LastError
Arguments:	None.
Return:	DECTError - Error code from the last API call.
Description:	This function returns the last error when communication with the shield. Its value is changed every time an API function is called and if there is no error the returned value is 0 (NoError).

Code Example:

```
void setup() {
  // ...
  dect.SetConfigs(&sConfigs);
  if (dect.LastError() != NoError) {
    // Communication error
  }
  // ...
}
```

Function:	Version
Arguments:	None.
Return:	stVersion - Version structure with the current firmware type and version.
Description:	This function is used to obtain the current shield version and to determine if it is working as a Portable or Fixed Part.

Code Example:

```
void setup() {
  // ...
  stVersion sVersion = dect.Version();
  if (sVersion.nFirmType == FP) {
    // Shield is working as Fixed Part
  } else {
    // Shield is working as Portable Part
  }
  // ...
}
```

Function:	Reset
Arguments:	None.
Return:	Nothing.
Description:	Resets the shield firmware to the default configurations.
Remarks:	All previously applied configurations are discarded.

Function:	Defaults
Arguments:	None.
Return:	Nothing.
Description:	Resets the shield firmware to the default configurations and also resets the DECT module. This means that the registration list, the list of known parts and the pincode are erased and replaced with the original values.
Remarks:	All previously applied configurations are discarded.

Function:	`GetConfigs`
Arguments:	`*sConfigs` - Pointer to a `stConfigs` structure to hold configurations.
Return:	If the function fails then it return `false`, otherwise `true`.
Description:	This function returns the current module configurations and blocks until the shield sends a response back which means that this function should always be the first to be called during setup.

Code Example:

```
void setup() {
  // ...
  stConfigs sConfigs;

  // Wait until the module is ready
  dect.GetConfigs(&sConfigs);

  // Verify if speaker is enabled
  if (sConfigs.sAudio.sSpeaker.bEnabled == true) {
    // Speaker is enable!
  }
  // ...
}
```

Remarks:	This is the only function in the API that blocks until an answer from the shield is received.

Function:	`SetConfigs`
Arguments:	`*sConfigs` - Pointer to a `stConfigs` structure with the configurations to apply.
Return:	If the function fails then it return `false`, otherwise `true`.
Description:	This function applies shield configurations from a previously modified `stConfigs` structure. It is highly recommended to obtain the current shield configurations, with the GetConfigs function, before modifying the structure.

Code Example:

```
void setup() {
  // ...
  stConfigs sConfigs;

  // Obtain current shield configurations
  dect.GetConfigs(&sConfigs);

  // Disable Microphone
  sConfigs.sAudio.sMic.bEnabled = false;

  // Enabled Loudspeaker
  sConfigs.sAudio.sSpeaker.bLoudspeakerEnabled = true;

  // Apply configurations
  dect.SetConfigs(&sConfigs);
  // ...
}
```

Function:	`GetStatus`
Arguments:	None.
Return:	`stStatus` structure containing the current shield status.
Description:	The function returns the current shield status information inside a `stStatus` structure. Please consult the contents inside the `stStatus` structure.

Code Example:

```
void loop() {
  // ...
  // Get shield status
  stStatus sStatus = dect.GetStatus();

  // Verifies if there is a file playing
  if (sStatus.sPlayback.bPlaying == true) {
    // There is a file currently playing!
  }
  // ...
}
```

D.4 DECT GROUP FUNCTIONS

This group includes the following functions:

- `stInfo GetInfo(void);`
- `stPincode GetPincode(void);`
- `void SetPincode(stPincode *pPinCode);`
- `void SetPincode(uint8_t n1, uint8_t n2, uint8_t n3, uint8_t n4);`
- `void SetPincode(uint8_t n1, uint8_t n2, uint8_t n3, uint8_t n4, uint8_t n5, uint8_t n6, uint8_t n7, uint8_t n8);`
- `uint8_t GetList(stListElem *pList, uint8_t nElements);`
- `bool Send(uint8_t nIdentifier, void *pData, uint8_t nSize);`
- `uint8_t Recv(uint8_t *pIdentifier, uint8_t *pBuffer, uint8_t nSize);`
- `bool Call(uint8_t nIdentifier, enum CallAction eAction);`
- `uint8_t GetCallList(stCallElem *pList, uint8_t nElements);`
- `bool Register(uint8_t nIdentifier);`
- `bool Scan(bool bEnable);`
- `uint8_t GetScanList(stScanElem *pList, uint8_t nElements);`
- `stDECTStatus GetDECTStatus(void);`

The description of each structure and function present in this group follows.

Structure:	`struct stInfo`
Description:	DECT Identifier structure.

```
typedef struct {
  union {
    uint8_t nRFPI[5]; // Radio Fixed Part Identifier
    uint8_t nIPUI[5]; // International Portable User Identity
  };
} stInfo;
```

Remarks:	The Fixed Part firmware uses the RFPI and the Portable Part uses the IPUI.

Structure:	`struct stPincode`
Description:	DECT Pincode structure.

```
typedef struct {
       uint8_t n1 :4; // 1st digit
       uint8_t n2 :4; // 2nd digit
       uint8_t n3 :4; // 3rd digit
       uint8_t n4 :4; // 4th digit
       // Optional
       uint8_t n5 :4; // 5th digit
       uint8_t n6 :4; // 6th digit
       uint8_t n7 :4; // 7th digit
       uint8_t n8 :4; // 8th digit
} stPincode;
```

Remarks:	Fields `n5` through `n8` are not used if their values are 0xF, which means the pincode will have 4 digits.

Structure:	`struct stListElem`
Description:	Element from a list of registered parts.

```
typedef struct {
  uint8_t nIdentifier; // The Identifier used to reference the part
  union {
    uint8_t nRFPI[5];  // Radio Fixed Part Identifier
    uint8_t nIPUI[5];  // International Portable User Identity
  };
} stListElem;
```

Remarks:	The Fixed Part firmware uses the RFPI and the Portable Part uses the IPUI.

Structure:	`enum CallAction`
Description:	Enumerator used to execute a call action.

```
enum CallAction {
  MakeCall = 0,     // Make a new call
  AcceptCall,       // Accept an incoming call
  RefuseCall = 2,   // Refuse call
  TerminateCall = 2 // Same as RefuseCall
};
```

Remarks:	Used exclusively with the `Call` function.

Structure:	`struct stCallElem`
Description:	Element from a list of calls.

```
typedef struct {
  enum CallState eState :3; // State of the call
  uint8_t bOutgoing     :1; // Call is outgoing
  uint8_t nLink         :4; // Link number associated with the call
} stCallElem;
```

Remarks:	The `CallState` enumerator is described below.

Structure:	`enum CallState`
Description:	Enumerator used to indicate a state of a call.

```
enum CallState {
  NoCall = 0,           // No call as yet been made
  PendingCall,          // Current call is pending
  OngoingCall,          // Call undergoing
  RefusedCall,          // Call was refused
  TerminatedCall,       // Call was terminated by user
  DestinationEndedCall, // Call was terminated by destination
  DisconnectedCall,     // Call was disconnect for some reason
  InvalidCall           // Call wasn't allowed to go through
};
```

Remarks:	Used in the list of calls to indicate the state of each call.

Structure:	`struct stScanElem`
Description:	Element from a list of scanned parts.

```
typedef struct {
  uint8_t nRSSI;    // Radio Signal Strength Indicator
  uint8_t nRFPI[5]; // Radio Fixed Part Identifier
} stScanElem;
```

Remarks:	There is no IPUI since no Portable Parts can be scanned.

Function:	`GetInfo`
Arguments:	None.
Return:	`stInfo` structure with the current DECT Identifier.
Description:	This function returns the DECT RFPI, if working as fixed part, or IPUI, if as portable part.

Code Example:

```
void loop() {
  // ...
  // Get shield IPUI and print it
  stInfo sInfo = dect.GetInfo();
  Serial.println(sInfo.nIPUI, DEC);
  // ...
}
```

Function:	`GetPincode`
Arguments	None.
Return:	`stPincode` structure with the current pincode discriminated.
Description:	This function returns a structure with the shield current pincode.
Remarks:	If the values of fields `n5` through `n8` in `stPincode` are 0x0F then the pincode has only 4 digits.

Function:	`SetPincode`
Arguments (1):	`*pPinCode` - `stPincode` structure with the new pincode to be applied to the shield.
Arguments (2):	`n1` - First digit of the pincode from 0 to 9; `n2` - Second digit of the pincode from 0 to 9; `n3` - Third digit of the pincode from 0 to 9; `n4` - Fourth digit of the pincode from 0 to 9.
Arguments (3):	`n1` - First digit of the pincode from 0 to 9; `n2` - Second digit of the pincode from 0 to 9; `n3` - Third digit of the pincode from 0 to 9; `n4` - Fourth digit of the pincode from 0 to 9; `n5` - Fifth digit of the pincode from 0 to 9; `n6` - Sixth digit of the pincode from 0 to 9; `n7` - Seventh digit of the pincode from 0 to 9; `n8` - Eighth digit of the pincode from 0 to 9.
Return:	Nothing.
Description:	All three variants of the `SetPincode` function change the pincode in the shield. The pin can have 4 or 8 digits, that is why there are two similar functions.

Function:	`GetList`
Arguments:	`*pList` - Pointer to a pre-allocated vector of `stListElem` structures; `nElements` - Number of elements in the `pList` vector.
Return:	Returns the number of elements written in the vector.
Description:	This function fills the `pList` vector with the list of known associated parts. These can be Portable Parts if the shield is working as a Fixed Part, or Fixed Parts if the shield is working as a Portable Part.

Code Example:

```
// Note: Function 'p' is a serial printf
void loop() {
  // ...
  uint8_t i, j;
  stListElem sList[10];

  i = dect.GetList(sList, 10);
  if (!dect.LastError()) {
    // Print header
    Serial.println(" Index |    IPUI/RFPI");
    if (i == 0)
      Serial.println("No parts registered!"));
    else
      // Print obtained parts information
      for (j = 0; j < i; j++)
        p("   %d     %02X %02X %02X %02X %02X\n\r", sList[j].nIndex,
          sList[j].nIPUI[0], sList[j].nIPUI[1], sList[j].nIPUI[2],
          sList[j].nIPUI[3], sList[j].nIPUI[4]);
  } else
    Serial.println("Failed");
  // ...
}
```

Remarks:	If the returned value is larger than the maximum number of elements, then the shield returned more elements and some didn't fit in the vector.

Function:	`Send`
Arguments:	`nIdentifier` - Shield identifier to where to send the data; `*pData` - Pointer to a byte array containing the data to send; `nSize` - The size of the data array.
Return:	If the function fails then it return `false`, otherwise `true`.
Description:	This function sends a data packet with up to 54 bytes to another shield using DECT. The `nIdentifier` argument identifies the target shield.
Remarks:	`nIdentifier` can be obtained from the `GetList` function.

Function:	`Recv`
Arguments:	`*pIdentifier` - Pointer to a pre-allocated byte to write the sender identifier; `*pData` - Pointer to a byte array to write the received data packet; `nSize` - The maximum size of the data array.
Return:	The total size of bytes received is returned.
Description:	This function can be called when there is a received data packet in the shield in order to transfer it to the Arduino.

Function:	`Call`
Arguments:	`nIdentifier` - Shield identifier/Call channel to where to send call action; `eAction` - Call action to be executed.
Return:	If the function fails then it return `false`, otherwise `true`.
Description:	This function executes a call action on a target shield. It can used to make, refuse or terminate a call for example.
Code Example:	

```
void loop() {
  // ...
  // Call all known Portable Parts
  dect.Call(0, MakeCall);
  // ...
}
```

Remarks:	The identifier 0 corresponds to all registered parts. Call channels can be obtained using the `GetCallList` function.

Function:	GetCallList
Arguments:	*pList - Pointer to a pre-allocated vector of stCallElem structures; nElements - Number of elements in the pList vector.
Return:	Returns the number of elements written in the vector.
Description:	This function fills the pList vector with the list of all call channels and their correspondent state. The call identifiers can also be used In the Call function to handle ongoing calls.

Code Example:

```
// Note: Function 'p' is a serial printf
void loop() {
  // ...
  uint8_t i, j;
  stCallElem sList[8];

  i = dect.GetCallList(sList, 8);
  if (!dect.LastError()) {
    // Print header
    p_P(PSTR(" Index | Link | Out | State\n\r"));
    if (i == 0)
      p_P(PSTR("No calls to inform!\n\r"));
    else
      // Print call state
      for (j = 0; j < i; j++) {
        p("   %d       %d    %s   ", j, sList[j].nLink, sList[j].bOutgoing?"Yes":"
No");
        if (sList[j].eState == NoCall) Serial.print("No Call");
        else if (sList[j].eState == PendingCall) Serial.print("Pending");
        else if (sList[j].eState == OngoingCall) Serial.print("Ongoing");
        else if (sList[j].eState == RefusedCall) Serial.print("Refused");
        else if (sList[j].eState == TerminatedCall) Serial.print("Terminated by
user");
        else if (sList[j].eState == DestinationEndedCall) Serial.print("Terminated by
destination");
        else if (sList[j].eState == DisconnectedCall) Serial.print("Disconnected");
        else if (sList[j].eState == InvalidCall) Serial.print("Invalid");
      }
  } else
    Serial.println("Failed");
  // ...
}
```

Remarks:	If the returned value is larger than the maximum number of elements, then the shield returned more elements and some didn't fit in the vector.

Function:	Register
Arguments:	nIdentifier - Identifier for the target shield to register;
Return:	If the function fails then it return false, otherwise true.
Description:	This function has a dual functionality: Fixed Part: nIdentifier is true or false and the function enables or disables registration; Portable Part: The shield will attempt to register to a target shield referenced by the nIdentifier value.

Function:	`Scan`
Arguments:	`bEnable` - Enable or disable scan;
Return:	If the function fails then it return `false`, otherwise `true`.
Description:	This function only works when the shield is working as a Portable Part and it enables or disables the registration mode.
Remarks:	The Portable Part has to be in scan mode in order to be able to register.

Function:	`GetScanList`
Arguments:	`*pList` - Pointer to a pre-allocated vector of `stScanElem` structures; `nElements` - Number of elements in the `pList` vector.
Return:	Returns the number of elements written in the vector.
Description:	This function fills the `pList` vector with the list of detected Fixed Parts in range with their respective identifier and RSSI. This function only works when the shield is working as a Portable Part.

Code Example:

```
// Note: Function 'p' is a serial printf
void loop() {
  // ...
  uint8_t i, j, k;
  stScanElem sList[10];

  i = dect.GetScanList(sList, 10);
  if (!dect.LastError()) {
    // Print header
    Serial.println(" Index | RSSI |      RFPI");
    if (i == 0)
      Serial.println("No fixed parts detected!");
    else
      // Print detected fixed parts
      for (j = 0; j < i; j++)
        p("   %d      %d/8   %02X %02X %02X %02X %02X\n\r", j, sList[j].nRSSI,
          sList[j].nRFPI[0], sList[j].nRFPI[1], sList[j].nRFPI[2], sList[j].nRFPI[3],
sList[j].nRFPI[4]);
  } else
    Serial.println("Failed");
  // ...
}
```

Remarks:	If the returned value is larger than the maximum number of elements, then the shield returned more elements and some didn't fit in the vector.

Function:	GetDECTStatus
Arguments:	None.
Return:	stDECTStatus structure with the current DECT status.
Description:	This function returns the shield's DECT status. This information can also be obtained using the GetStatus function.

Code Example:

```
void loop() {
  // ...
  // Get shield DECT status
  stDECTStatus sStatus = dect.GetDECTStatus();
  // Prints the number of available data packets
  Serial.println(sStatus.sData.nDataPackets, DEC);
  // ...
}
```

D.5 PLAYBACK GROUP FUNCTION

This group includes the following functions:

- `void StartPlaying(void);`
- `void StopPlaying(void);`
- `void SetPlayVolume(uint8_t nVolume);`
- `stPlaybackStatus GetPlaybackStatus(void);`
- `bool OpenFile(const char *pFilePath);`
- `bool SetPlaybackQueue(uint8_t nSize);`
- `bool QueueFile(const char *pFilePath);`

The description of each structure and function present in this group follows.

Function:	StartPlaying
Arguments:	None.
Return:	Nothing.
Description:	This function plays a previously opened file, or a sequence of files, in the module speaker and loudspeaker if enabled.
Remarks:	An audio file must be opened with OpenFile first.

Function:	StopPlaying
Arguments:	None.
Return:	Nothing.
Description:	If there is a file playing, this function will sever the audio stream and stop the playback.

Function:	SetPlayVolume
Arguments:	nVolume - Playback volume.
Return:	Nothing.
Description:	This function changes the file playback volume (not the speaker) from 0 (muted) to 15 (maximum).

Code Example:

```
void loop() {
  // ...
  // Set the playback volume to the maximum value
  dect.SetPlayVolume(15);
  // ...
}
```

Function:	GetPlaybackStatus
Arguments:	None.
Return:	stPlaybackStatus structure with the current playback status.
Description:	This function returns the shield's playback status. This information can also be obtained using the GetStatus function.

Code Example:

```
void loop() {
  // ...
  // Get shield playback status
  stPlaybackStatus sStatus = dect.GetPlaybackStatus();
  // Verifies if the previously opened file is valid
  if (sStatus.bValid == true) {
    // File is valid!
  }
  // ...
}
```

Function:	OpenFile
Arguments:	*pFilePath - String with the file path and name in the memory card to open.
Return:	If the function fails then it return false, otherwise true.
Description:	This function opens an existing audio file from the memory card. This function always resets the playback queue size to 1.

Code Example:

```
void loop() {
  // ...
  // Open file and play it
  dect.OpenFile("music.wav");
  dect.StartPlaying();
  // ...
}
```

Remarks:	Current firmware version only supports 8 kHz WAV files with 8-bit μ-law encoding.

Function:	`SetPlaybackQueue`
Arguments:	`nSize` - New size of the playback queue.
Return:	If the function fails then it return `false`, otherwise `true`.
Description:	This function changes the size of the playback queue. Its size represents the maximum number of files that can be loaded and played in sequence.
Remarks:	The shield will only set the intended size if it was enough memory to do so.

Function:	`QueueFile`
Arguments:	`*pFilePath` - String with the file path and name in the memory card to open.
Return:	If the function fails then it return `false`, otherwise `true`.
Description:	This function opens an existing audio file from the memory card and places it on the bottom of the playback queue. If the queue is full or the file is invalid, the function will return `false`.

Code Example:

```
void loop() {
  // ...
  // Opens 2 files and plays them in sequence
  dect.SetPlaybackQueue(2);
  dect.QueueFile("music1.wav");
  dect.QueueFile("music2.wav");
  dect.StartPlaying();
  // ...
}
```

Remarks:	Current firmware version only supports 8 kHz WAV files with 8-bit μ-law encoding.

D.6 AUDIO GROUP FUNCTIONS

This group includes the following functions:

- `void EnableSpeaker(void);`
- `void DisableSpeaker(void);`
- `void EnableLoudspeaker(void);`
- `void DisableLoudspeaker(void);`
- `void SetSpeakerVolume(uint8_t nVolume);`
- `void EnableMicrophone(void);`
- `void DisableMicrophone(void);`
- `void SetMicrophoneGain(uint8_t nGain);`
- `stAudioConfigs GetAudioConfigs(void);`

The description of each structure and function present in this group follows.

Function:	`EnableSpeaker`
Arguments:	None.
Return:	Nothing.
Description:	This function enables the shield speaker output.

Function:	`DisableSpeaker`
Arguments:	None.
Return:	Nothing.
Description:	This function disables the shield speaker output.

Function:	`EnableLoudspeaker`
Arguments:	None.
Return:	Nothing.
Description:	This function enables the shield loudspeaker output.

Function:	`DisableLoudspeaker`
Arguments:	None.
Return:	Nothing.
Description:	This function disables the shield loudspeaker output.

Function:	`SetSpeakerVolume`
Arguments:	`nVolume` - Speaker attenuation from 0 to 7.
Return:	Nothing.
Description:	This function changes the speaker and loudspeaker attenuation from 0 (-2dB) to 7 (12dB).

Code Example:

```
void loop() {
  // ...
  // Set the speaker attenuation to -2dB (maximum volume)
  dect.SetSpeakerVolume(0);
  // ...
}
```

Function:	`EnableMicrophone`
Arguments:	None.
Return:	Nothing.
Description:	This function enables the shield microphone input.

Function:	`DisableMicrophone`
Arguments:	None.
Return:	Nothing.
Description:	This function disables the shield microphone input.

Function:	SetMicrophoneGain
Arguments:	nGain - Microphone gain from 0 to 15.
Return:	Nothing.
Description:	This function changes the microphone gain from 0 (0dB) to 15 (30dB).

Code Example:

```
void loop() {
  // ...
  // Set the microphone gain to its maximum sensitivity of 30dB
  dect.SetMicrophoneGain(15);
  // ...
}
```

Function:	GetAudioConfigs
Arguments:	None.
Return:	stAudioConfigs structure with the current audio configurations.
Description:	This function returns the shield audio configurations. This information can also be obtained using the GetConfigs function.

Code Example:

```
void loop() {
  // ...
  // Get shield audio configurations
  stAudioConfigs sConfigs = dect.GetAudioConfigs();
  // Prints the current speaker volume
  Serial.println(sConfigs.sSpeaker.nVolume, DEC);
  // ...
}
```

D.7 MICROSD GROUP FUNCTIONS

This group includes the following functions:

- `void AccessmicroSD(void);`
- `void ReleasemicroSD(void);`

The description of each structure and function present in this group follows.

Function:	AccessmicroSD
Arguments:	None.
Return:	Nothing.
Description:	This function enables to Arduino to access the shield's microSD slot.
Remarks:	Other shields can't use the SPI while the shield is granting access to the Arduino.

Function:	`ReleasemicroSD`
Arguments:	None.
Return:	Nothing.
Description:	This function releases the microSD card from Arduino control. This function should always be called when the Arduino doesn't need to access the card anymore.
Remarks:	Other shields can use the SPI bus after calling this function.

D.8 EVENT GROUP FUNCTION

This group includes the following functions:

- `bool GetEvent(stEvent *pEvent);`
- `void attachRecvCallback(void (*recv)(uint8_t, uint8_t*, uint8_t));`
- `void attachCallCallback(void (*call)(uint8_t, enum CallState, enum CallState));`
- `void attachPlaybackStoppedCallback(void (*pbstop)(void));`
- `void attachPlaybackCallback(void (*pbnext)(uint8_t));`

In order to enable events, it is necessary to configure the shield using the `SetConfigs` function as exemplified in Figure D.7.

Figure D.7 - Event configuration code.

```
void setup() {
  // ...
  stConfigs sConfigs;
  dect.GetConfigs(&sConfigs);
  // Enable Data Events
  sConfigs.sEvents.bDataEvents = true;
  // Enable Call Events
  sConfigs.sEvents.bCallEvents = true;
  // Enable Playback Events
  sConfigs.sEvents.bPlaybackEvents = true;
  dect.SetConfigs(&sConfigs);
  // ...
}
```

The description of each structure and function present in this group follows.

Structure:	`struct stEvent`
Description:	Event structure.

```
typedef struct {
  enum EventType eEventType :8; // Event Type
  uint8_t nData[2];             // Date related to the event
} stEvent;
```

Remarks:	The `enum EventType` enumerator is described below.

Structure:	`enum EventType`
Description:	Event type.

```
enum EventType {
  NoEvent = 0,          // No event
  DataEvent,            // Data packed available
  CallEvent,            // Call changed status
  StoppedPlayingEvent, // Playback stopped
  NextFileEvent,        // Next file started playing
};
```

Function:	`GetEvent`
Arguments:	`*pEvent` - Pointer to a pre-allocated `stEvent` structure to hold an event.
Return:	If the function fails then it return `false`, otherwise `true`.
Description:	This function saves the oldest pending event from the shield inside the `pEvent` argument.
Remarks:	Some events (e.g. receive event) aren't removed until their associated information is processed (e.g. read the incoming packet).

Function:	`attachRecvCallback`
Arguments:	`*recv` - Pointer to a callback function.
Return:	Nothing.
Description:	This function attaches a callback to the event triggered when there is a new incoming data packet available in the shield. See example below for more information.

Code Example:

```
void recvCallback(uint8_t nIdentifier, uint8_t *pBuffer, uint8_t nSize)
{
  // New data packet received
  // nIdentifier - Sender identification
  // *pBuffer - Buffer with the received data
  // nSize - Number of bytes present in the buffer
  // TODO: Add processing code here
}

void setup() {
  // ...
  // Set callback for incoming data packets
  dect.attachRecvCallback(recvCallback);
  // ...
}
```

Remarks:	Event needs to be enabled using the `SetConfigs` function. See the `stEventsConfigs` structure.

Function:	attachCallCallback
Arguments:	*call - Pointer to a callback function.
Return:	Nothing.
Description:	This function attaches a callback to the event triggered when there is a status change on one call. See example below for more information.

Code Example:

```
void callCallback(uint8_t nIdentifier, enum CallState eNewState, enum CallState
eOldState)
{
  // Call changed state
  // nIdentifier - Call Identifier
  // eNewState - New state of the call
  // eOldState - Old state of the call
  // TODO: Add processing code here
}

void setup() {
  // ...
  // Set callback for when a call status changes
  dect.attachCallCallback(callCallback);
  // ...
}
```

Remarks:	Event needs to be enabled using the SetConfigs function. See the stEventsConfigs structure.

Function:	attachPlaybackStoppedCallback
Arguments:	*pbstop - Pointer to a callback function.
Return:	Nothing.
Description:	This function attaches a callback to the event triggered when the shield stops playing a file or sequence. See example below for more information.

Code Example:

```
void pbstopCallback(void)
{
  // Playback stopped
  // TODO: Add processing code here
}

void setup() {
  // ...
  // Set callback for when the playback stops
  dect.attachPlaybackStoppedCallback(pbstopCallback);
  // ...
}
```

Remarks:	Event needs to be enabled using the SetConfigs function. See the stEventsConfigs structure.

Function:	`attachPlaybackNextCallback`
Arguments:	`*pbstop` - Pointer to a callback function.
Return:	Nothing.
Description:	This function attaches a callback to the event triggered when the shield starts playing the next file in the sequence. See example below for more information.

Code Example:

```
void pbnextCallback(uint8_t nNext)
{
  // Playback started playing the next file
  // nNext - Number of the current file playing
  // TODO: Add processing code here
}

void setup() {
  // ...
  // Set callback for when the playback goes to the next file
  dect.attachPlaybackNextCallback(pbnextCallback);
  // ...
}
```

Remarks:	Event needs to be enabled using the `SetConfigs` function. See the `stEventsConfigs` structure.